国外经典计算机科学教材

computer systems

Computer Systems
A Programmer's Perspective

深入理解计算机系统

[美] Randal E. Bryant
David O'Hallaron 著
龚奕利 雷迎春 译

Randal E. Bryant and David O'Hallaron







Computer Systems A Programmer's Perspective

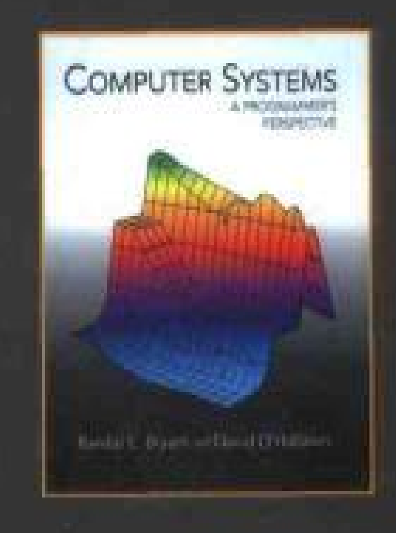
深入理解计算机系统

看看指导教师们是如何评价这本书的：

"我坚信从程序员的角度来看计算机系统对教会学生计算机的内部结构非常有帮助。"
——Kostas Daniilidis，宾夕法尼亚大学

"这本书讲述事物的方法与众不同，但是和我想要的课程进行方式类似。"
——John Greiner，Rice 大学

"这是一项出色的工作，是这一领域教学方法的一次革命。"
——Michael Scott，罗切斯特大学



从程序员的视角，看计算机系统

本书适合那些想要写出更快、更可靠程序的程序员阅读。通过掌握程序是如何映射到系统上，以及程序是如何执行的，读者能够更好地理解程序的行为为什么是这样的，以及效率低下是如何造成的。粗略来看，计算机系统包括处理器和存储器硬件、编译器、操作系统及网络互联环境。而通过程序员的视角，读者可以深深地体会到学习计算机系统的内部工作原理会对他们今后作为计算机科学研究者和工程师的工作有进一步的帮助。它还有助于为进一步学习计算机体系结构、操作系统、编译器和网络互联打下基础。

本书的主要论题包括：数据表示、C程序的机器级表示、处理器结构、程序优化、存储器层次结构、链接、异常控制流、虚拟存储器和存储器管理、系统级I/O、网络编程和并发编程。书中所覆盖的内容主要讲述这些方面是如何影响应用和系统程序员的。例如，在讲述数据表示时，本书提出了用来表示数字的表示方法是有限的，它能够近似地表示整数和实数，但是这种表示方法是有限制的，程序员必须了解。在讲述高速缓存时，本书讨论了矩阵代码中的循环变量的顺序是如何影响程序的性能的。在讨论网络互联时，本书描述了并发服务器如何有效地处理来自多个客户端的请求。

本书基于Intel兼容（IA32）机器，在Unix或者相关的操作系统（例如Linux）上执行C程序。虽然书中包括了一些帮助读者将Java转化成C的提示，但还是要求读者对C或者C++有一定的了解。



Randal E. Bryant 现任卡耐基梅隆大学计算机科学主任教授（President's Professor）和计算机学院院长。他从事本科和研究生计算机系统方面课程的教学超过20年。在讲授计算机体系结构课程多年后，他开始把关注点从如何设计计算机转移到程序员如何在更好的了解系统的情况下编写出更有效和更可靠的程序。Bryant教授研究成果丰硕，包括Intel、Motorola、IBM和Fujitsu在内的主要计算机制造商都使用过他的研究成果。他还因他的研究获得过数项大奖。他同时是ACM和IEEE的院士。



David R. O'Hallaron 是卡耐基梅隆大学计算机科学系及电子和计算机工程系的副教授。他教授计算机系统方面的课程，例如计算机体系结构、计算机系统导论、并行处理器设计和Internet服务。O'Hallaron教授获得了CMU计算机科学院颁发的Allen Newell优秀研究奖章。他为Quake项目创立的基准程序，183.equake，被选入非常有影响力的SPEC CPU和OMP（Open Mp）基准程序包中。

责任编辑 / 姚贵胜
封面设计 / 王红柳





www.PearsonEd.com

ISBN 7-5083-2175-8
定价：85.00 元

国外经典计算机科学教材

Computer Systems
A Programmer's Perspective

深入理解计算机系统

[美] Randal E. Bryant
David O'Hallaron 著
龚奕利 雷迎春 译

中国电力出版社
www.infopower.com.cn

Computer Systems: A Programmer's Perspective (ISBN 0-13-034074-X)
Randal E. Bryant, David R. O'Hallaron

图书在版编目（CIP）数据
深入理解计算机系统 / （美）布赖恩特（Bryant,R.E.）等著；龚奕利，雷迎春译．—北京：
中国电力出版社，2004
（国外经典计算机科学教材系列）
ISBN 7-5083-2175-8
Ⅰ.深… Ⅱ.①布…②龚…③雷… Ⅲ.计算机系统－高等学校－教材 Ⅳ.TP30
中国版本图书馆 CIP 数据核字（2004）第 014328 号

丛 书 名：国外经典计算机科学教材系列
书　　名：深入理解计算机系统
编　　著：（美）Randal E. Bryant, David R. O'Hallaron
翻　　译：龚奕利　雷迎春
责任编辑：姚贵胜
出版发行：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092 1/16　印　张：53.25　字　数：1209千字
书　　号：ISBN 7-5083-2175-8
版　　次：2004 年 5 月北京第 1 版　2004 年 5 月第 1 次印刷
定　　价：85.00 元

出版说明

新世纪的朝阳刚刚露出丝抹微红，如火如荼的全球信息化浪潮便汹涌而至，让人无时无刻不感受到新一轮产业革命的气息。如何在这场变革中占尽先机，既是对民族信息业的挑战，也是机遇。从而，作为民族信息产业发展基石的高等教育事业就被赋予了比以往更重的责任，对培养和造就我国21世纪的一代新人提出了更高的要求。但在计算机科学突飞猛进的同时，专业教材的发展却严重滞后，越来越成为人才培养的瓶颈。同时，以美国为代表的西方国家计算机科学教育经历了充分的发展，产生了一批有着巨大影响力的经典教材，因此，以批判、借鉴的态度有选择地引进这些国外经典计算机教材，将促进国内教学体系和国外接轨，大大推动我国计算机教育事业的发展

中国电力出版社进入计算机图书市场已有近6个年头，通过坚持"高端、精品、经典"战略，致力于与国外著名出版机构合作，出版了大批博得计算机业界和教育界赞誉的作品。通过与信息技术教育界人士的广泛沟通，同时依托丰富的出版资源，中国电力出版社适时推出了"国外经典计算机科学教材"的出版计划。本次教材出版计划是和美国最大的计算机教育出版机构——Pearson教育集团（Addison-Wesley、Prentice-Hall等皆为其下属子公司）合作，依托其数十年积累的大批经典教材资源，确保了教材选题的权威经典。

为保证这套教材的含金量，并做到有的放矢，我们在国内组织了由中国科学院、北京大学等一流院校教师组成的专家指导委员会，对高校课程教学体系做了系统、详细的调查，听取了众多教育专家、行业专家的意见，对教育部的教育规划进行了认真研究，并深入了解国外大学实际教学选用的教材状况，对国外教材做了理性的分析，确立了依托国家教育计划、传播先进教学理念、为培养符合社会需要的高素质创新型人才服务，来作为本次"国外经典计算机科学教材"出版计划的宗旨。

我们从2002年的下半年开始着手这套教材的策划工作，并多次组织了专家研讨会、座谈会等，分析现有教材的优点与不足，采其精华，并力争体现本套教材的质量和特色。

1. 深入理解国内的教学体系结构，并比照国外相同专业的课程设置，既具有现实的适用性，又立足发展眼光，具备一定的前瞻性。

2. 以计算机专业的核心课程为基础，同时配合专业教学计划，争取覆盖专业选修课程和专业任选课程。

3. 选取国外的最新教材版本，同时对照国内同专业课程的学时要求，对不适用的版本进行剔除，充分满足国内教学要求。

4. 根据专业对口和必须具备同课程教学经验的要求，严格挑选译者，并严把质量关，确保教材翻译的高质量。

5. 通过从原出版社网站下载勘误表及与原书作者进行沟通的方式，对原书中的错误一一做了修改。

6. 对教材出版的后期工作，如审校、编辑、排版、印刷进行了严格的质量把关。

经过专家指导委员会的集体讨论，并广泛听取广大高等院校师生的意见，反复比较，从数百种国外教材中遴选出数十种，列入第一阶段的出版计划。这些教材的作者无一不是学富五车的大师，如 Stallings，Date，Ullman，Aho，Bryant，Sedgewick 等，他们的作品均是一版再版，并被众多国外一流大学如 Stanford University，MIT，UC Bekerley，Carnegie Mellon Univeristy，University of Michigan 等采用为教材。拟订的第一阶段出版计划包括 30 种图书，内容覆盖程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等计算机专业核心基础课程，基本满足国内计算机专业的教学要求。

此外，为了帮助广大任课教师加深对本系列教材的理解，减轻他们的备课难度，我们从国外出版机构引进了大批的课程教学辅助资料，并积极延请国内优秀教师，根据其使用该系列教材中的教学经验，着手编写更加适合国内应用状况的教辅材料。

由于我们对国内高校计算机教育存在认识深度上的不足，在选题、翻译、编辑加工出版等方面的工作中还有许多有待提高之处，恳请广大师生和读者提出批评和建议，并期待有更多的人加入到我们的工作中来。我们的联系方式是：

电子邮件：csbook@cepp.com.cn

联系电话：010-88515918-300

联系地址：北京市西城区三里河路 6 号中国电力出版社

邮政编码：100044

译　序

作为一个程序员，我们经常被一些奇怪的程序问题所困扰。例如最近，我的一位朋友从经典的数据结构书上抄了一段关于计算有向图的函数实现代码，这个函数实现在 gcc 环境下的编译一点问题都没有，但只要一实际运行，就会报段错误。我看了这段代码后，立刻就意识到问题出现在哪里了——他在函数实现里分配了一个 16MB 的局部变量导致栈溢出。类似这样的问题，还会有许多。不过在这些问题中，让程序员有麻烦的已经不是编程语言本身的问题，而是需要程序员更好地理解计算机系统，知道程序如何在计算机上被执行。理解计算机系统，不是简单地从书市上购买一些介绍计算机系统的书，读一读而已。迄今为止，我对市面上这类书的了解是：对于大多数程序员而言它们都过于专业化，且从书的内容和语言组织上都偏重于原理的介绍，一般程序员很难有时间和精力去消化和吸收书中的内容，更无从用这些计算机系统的知识来帮助自己解决程序问题。

事实上，高级语言编程和计算机系统被编程环境如 gcc 划分成两张皮，尽管程序员能用高级语言驱动计算机完成指定的计算任务，可是却不一定能很清楚地知道计算机是如何解释和执行程序代码的。

我本人是一个计算机专业科班出身的人，学生期间学习到许多关于计算机系统的知识。可在实际研究工作中，过去所学的计算机系统知识变得遥远和模糊。1999 年初，出于研究兴趣的目的，我设计了一个高性能网络服务器结构，并编写了它的实现。在此期间，使我明白一个高性能服务器程序与计算机系统之间的唇齿相依的关系。过去，促使我建立高级语言和计算机系统的联系来主要自于研究的压力，其采用的方法是遍寻国外关于系统编程的邮件列表，并结合以往所学的计算机系统知识。这种方法固然能帮助我解决实际所碰到的问题，但却需要花费大量的时间且没有条理。2003 年元月，编辑部让我帮助他们从一批刚出版的外文书中挑一些可以在国内推广的书，我一眼就看中了这本由 Bryant 和 O'Hallaron 所著的《Computer Systems: A Programmer's Perspective》，它就是我过去想要的书，我相信也是每一个想了解计算机系统的程序员想要的书。我迫不及待地从编辑手中抢下此书的翻译工作，这个临时添加的任务改变了我和另一位译者 2003 年的生活。2003 年 8 月底，终于完成此书的翻译工作，并起中文名为《深入理解计算机系统》。

《深入理解计算机系统》的最大优点是为程序员描述计算机系统的实现细节，帮助其在大脑中构造一个层次型的计算机系统，从最底层的数据在内存中的表示（如大多数程序员一直陌生或疑惑的浮点数表示），到流水线指令的构成，到虚拟存储器，到编译系统，到动态加载库，到最后的用户态应用。贯串本书的一条主线是使程序员在设计程序时，能充分意识到计算机系统的重要性，建立起被所写程序可能被执行的数据或指令流图，明白当程序被执行时，到底发生了什么事。从而能设计出一个高效、可移植、健壮的程序，并能够更快地对程序排错、调整程序性能等。

本书是通过程序员的视角来介绍计算机系统，即首先把高级语言转换成计算机所能理解的一种中间格式（如汇编语言），然后描述计算机如何解释和执行这些中间格式的程序，是系统的哪一部分影响程序的执行效率。所以，在讲述计算机系统知识的同时，也顺便给出了关于 C 语言和汇编语言

（有可能是编译系统产生的）的编程和阅读技巧，以及基本的系统编程技巧和工具，同时，还给出一些方法帮助程序员基于对计算机系统的理解来度量和改善程序的性能、及其它棘手问题。

本书的主要内容是关于计算机体系结构（高级硬件设计）与编译器和操作系统的交互，包括：数据表示；汇编语言和汇编级计算机体系结构；处理器设计；程序的性能度量和优化；程序的加载器、链接器和编译器；包括 I/O 和设备的存储器层次结构；虚拟存储器；外部存储管理；中断、信号和进程控制。对这些不同领域知识的介绍使我们能在编写系统程序时，基于系统性能的考虑，采取一个更好的折中方案。

本书强调对计算机系统的概念的理解，但并不意味着不动手。如果按照本书的安排做每一章后面的习题，将有助于理解和加深正文所述的概念和知识，并且有时候，可以从实际动手中学习到新的知识。如果不动手，空洞地去看文字，是很难理解文字背后的意义的。我个人的经验是，有许多系统设计和概念，看似简单或不理解，可一旦自己动手做同样的试验，才更明白当初的设计者为什么要如此设计。计算机系统就像自然界的生态环境，对每一个部件的设计都要求它能融洽地和系统内其他部件和平相处，我们不能站在一个微观的视角去看待系统部件的设计是否最优，而应该从宏观来观察和思考。

为方便理解本书的内容，本书的读者假定具备 C 语言编程的能力。由于原书是卡内基梅隆大学（CMU）的教材，且被其他一些著名的大学也选用为教材，因此，本书的读者不仅仅是那些因为工作和兴趣而关注本书的人，还包括一些在校的大学生，作为他们的教材或辅助性资料。个人认为，在校学生越早接触本书的内容，将越有利于他们学习计算机的相关课程，培养对计算机系统的研究兴趣。

总而言之，《深入理解计算机系统》一书是一个桥梁，它帮助程序员衔接了计算机系统的各个领域的知识，为程序员构造了一个概念性框架。对于各个领域（如计算机系统结构、处理器、操作系统、编译器、网络、并发编程）的知识进一步获取，还需要参考相关书籍。

参加翻译的还有龚奕利、易金华和陈永兴等，在此也特别表示感谢。

由于此书的内容量大，加上翻译时间并不很宽裕，尽管我们十分努力，但还是难以避免出现错误，以及存在许多不尽人意的地方，欢迎广大读者批评指正，以便改进。

雷迎春

2004.2.15

于北京中关村（中科院）青年公寓

关于术语的翻译

本书跨越计算机的多个领域，涉及了许多专业的术语。在翻译的过程中，我们尽可能地忠实反映原文的意思，但并不是每个术语的翻译都那么恰当，符合每个读者的阅读习惯。不可避免地，对某些术语的翻译带了我们个人的习惯和偏好，希望读者谅解。下面，我们解释在本书中频繁出现的一些术语的翻译。

directive

这个单词多用来描述 C 语言中类似#include 的语句，或汇编语言中类似.pos 的语句。按照我的认识，这个单词应该译做"指令"比较恰当，起着指导或导引的作用。但是，在 directive 单词出现的地方，还同时出现了 instruction 单词（这种现象以第 3 章为主），其中文的含义也是"指令"。相比于 directive、instruction 显然是一个更强势的单词。为了从中文字面上区分这两个单词，方便读者阅读，我们在不影响基本意思的前提下，翻译 directive 为命令。

operation

这是一个遍布全书的单词。它众多的意思中有两个是："操作"和"运算"。对这个单词的翻译，我们没有采用一刀切的方法，而是尽可能采用国内读者的习惯来翻译。根据我自己的切身体会，以及网友对 operation 译法的讨论，我们更倾向于在数学的领域内使用"运算"，而在计算机领域使用"操作"。基于这个认识，以及章节内容的安排，我们把第 2 章（该章涉及大量的数学描述）中出现的 operation 主要译做"运算"，而把其他章节中的 operation 主要翻译为"操作"。需要注意的是，这种划分并不是绝对的。

memory 与 storage

memory 是一个我们非常熟悉的术语，我们一般把它习惯地称为"内存"。但是，通过本书第 6 章对 memory 的解释来看，仅有这样的理解是不足够的。本书认为 memory 可以是不同容量、成本和访问时间的存储设备，我们过去所认识的 memory 只是 DRAM。所以，不能把 memory 简单地翻译为"内存"。

从 memory 和 storage 这两个单词的中文意思来看，memory 是"存储器"，而 storage 是"存储，存储器"。另外，我们还观察到，memory 更多地以名词出现，描述一个静态的物理设备，而 storage 除了可以作为名词出现外，还有动词的形式（store、storing 和 stored）。所以，我们取 memory 的中文意思为"存储器"，而取 storage（以及 store、storing 和 stored）的中文意思为"存储"。除此之外，如果在一句话中，有 memory 和 storage 同时出现时，我们除了给出 storage 的中文释义外，还尽可能地附英文单词，以示与 memory 的区别。

hazard

这是一个很扰人的体系结构领域的术语。在本书中，我们选用它的中文释义为“冒险”。实际上，我们在做学生时，大都直呼它的英文，很少说它的中文，选择它的中文译法真的是一件很麻烦的事情。曾经有一个网友告诉我，他看到一个“险象”的译法比较贴切。呵呵，为了这个术语的翻译，我和他在网上争论了两天。仔细想想“险象”这种译法，确实不为错，但还不能完全说服我选用它。因为，这个释义太过陌生，许多读者可能无法联想到其对应的英文单词。而选用“冒险”，尽管不是那么完美，但是大多数研究体系结构的读者会很熟悉。所以，对“冒险”的选用只是一种习惯和默认。

timer

timer 的中文释义有：“定时器，计时器”。尽管这两个中文释义都可以描述一个现象：间隔一段时间后产生一个事件，但是我们认为这两者之间是有区别的。从中文字面来说，定时器的间隔更多是固定的，倾向于静态性；而计时器的间隔更多是不固定的，有计算的意思，倾向于动态性。所以，在本书的第 9 章中（该章主要描述系统评价），我们主要把 timer 翻译为计时器，而在其他章节翻译为定时器。

local

local 的中文释义是：“本地，局部”。我们没有严格地区分它，完全是根据上下文描述的方便，来选用不同的释义。

原书还出现了一些错误，这些错误只是我们个人认为的，很有可能是我们理解错了。所以，我们在“篡改”原文的意思时，还尽可能地给出了原文的意思，以帮助读者甄别。

我很喜欢这本书，且认为它的内容在 5~10 年内都有它存在的价值。但是，鉴于我们的能力和时间有限，不能保证完全忠实、准确地重述原文的意思，还需要广大读者的支持。希望广大读者在阅读本书的时候能积极地给我们指出其中错误，改善此书的质量，方便后来的读者从中更顺畅地获取知识。

前　言

《深入理解计算机系统》（Computer Systems: A Programmer's Perspective，CS: APP）这本书的主要读者是那些想通过学习计算机系统的内在运作而提高自身技能的程序员。

我们的目的是解释所有计算机系统的本质概念，并向你展示这些概念是如何实际地影响应用程序的正确性、性能和实用性的。与其他主要针对系统构造人员的系统类书籍不同，这本书是写给程序员的，是从程序员的角度来描述的。

如果你学习和研究这本书里的概念，你将步入稀缺的"权威程序员"的行列，将知道事情是如何运作的，也知道在出现故障时如何进行修复。同时，你也将做好学习其他具体系统主题的准备，比如编译器、计算机体系结构、操作系统、嵌入式系统和网络互联。

读者所应具备的背景知识

本书中的范例是基于英特尔兼容的处理器（英特尔称之为"IA32"，即俗称的"x86"）、在 Unix 或类 Unix（比如 Linux）操作系统上运行的 C 语言程序。（为了简化我们的描述，我们将用 Unix 统称 Solaris 和 Linux 这样的系统。）文中包括了大量已在 Linux 系统上编译和运行过的程序范例。我们假设你能访问一台这样的机器，并且能够登录，然后做一些诸如修改目录之类的简单操作。

如果你的计算机运行的是 Microsoft Windows 系统，你有两种选择。第一，获取一个 Linux 的拷贝（参见 www.linux.org 或者 www.redhat.com），然后以"双重启动"模式安装它，这样你的机器就能运行任一个操作系统了。另一种选择就是，通过安装 Cygwin 工具（www.Cygwin.com），你就能在 Windows 下得到一个类似 Unix 的 shell 以及一个非常类似于 Linux 提供的环境。不过，Cygwin 并不能提供所有的 Linux 功能。

我们还假设你对 C 和 C++有一定的了解。如果你以前只有 Java 经验，那么这种转换将需要你自己付出更多的努力，不过我们也将帮助你。Java 和 C 有相似的语法和控制语句。

但是，有一些 C 语言的内容，特别是指针、显式的动态存储器分配和格式化 I/O，Java 中都是没有的。所幸的是，C 是一个较小的语言，并且在 Brain Kernighan 和 Dennis Ritchie 经典的"K&R"文字中得到了清晰优美的描述[40]。无论你的编程背景如何，K&R 都应是你个人图书收藏的一部分。

这本书的前几章揭示了 C 语言程序和它们相应的机器语言程序之间的交互作用。机器语言示例都是用运行在 Intel IA32 处理器上的 GNU GCC 编译器生成的，我们不需要你以前有任何硬件、机器语言或是汇编语言编程的经验。

给 C 语言初学者：关于 C 编程语言的建议

为帮助 C 语言编程背景薄弱的（或全无背景的）读者，也为了强调 C 中一些重要特性，我们做了专门的注释。我们假设你熟悉 C++或 Java。

怎样阅读此书

从程序员的角度来学习计算机系统如何工作将非常有趣，主要是因为这个过程可以非常主动。无论何时你学到一些新的东西，都可以马上试验并且直接看到运行结果。事实上，我们相信学习系统的惟一方法就是做（do）系统，即在真正的系统上解决具体的问题，或是编写和运行程序。

这种主题观念贯穿全书。当引入一个新概念时，紧随其后的将是一个或多个练习题，你应该马上做一做来检验你的理解。练习题的解答在每章的末尾。当你阅读时，尝试自己来解答每个问题，然后再查阅答案，看看自己是否正确。每一章后面都有一组不同难度的家庭作业题。你的指导老师在教师手册中有这些问题的答案。对每个家庭作业题，我们标注了我们认为的难度级别：

◆只需要几分钟。几乎或完全不需要编程。

◆◆可能需要将近 20 分钟。通常包括编写和测试一些代码，许多都取自我们在考试中的题目。

◆◆◆需要很大的努力，也许是 1~2 个小时。一般包括编写和测试大量的代码。

◆◆◆◆一个实验作业，需要将近 10 个小时。

文中每段代码示例都是 C 程序，经过版本为 2.95.3 的 GCC 编译并在内核版本为 2.2.16 的 Linux 系统上测试后直接生成的，没有任何人为的改动。所有源程序代码均可从本书的主页（csapp.cs.cmu.edu）上获取。在文中，源程序的文件名列在两条水平线的右边，水平线之间是格式化代码。比如，图 P.1 中的程序能在 code/intro 目录下的 hello.c 文件中找到。我们鼓励你，当遇到这些示例程序时，在你的系统上试试运行它们。

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6  }
```

code/intro/hello.c

图 P.1 一个典型的代码示例

最后，有些部分（用“*”标注的）包含了一些你可能会觉得有趣但可以略过而不影响阅读连贯性的东西。

旁注：什么是旁注？

整本书中，你将会遇到很多以这种形式出现的旁注。旁注是附加说明，能使你对当前讨论的主题多一些了解。旁注有很多目的。一些是小的历史故事，例如，C、Linux 和 Internet 是从何而来的？有时，旁注是用来阐明学生们经常感到疑惑的问题，例如，高速缓存的行、组和块有什么区别？还有的时候，旁注给出了一些现实世界的例子，例如，一个浮点错误怎么毁掉了法国的一枚火箭，或是一个真正的 IBM 磁盘驱动器看上去是什么样子。最后，还有一些旁注仅仅就是笑料，例如，什么是“hoinky”？

把一个常量乘法转化为一系列的移位和加法。我们用 C 的位级操作来说明布尔代数的原理和应用。我们从如何表示浮点值和浮点操作的数学属性方面讲述 IEEE 标准的浮点格式。

对计算机算术的深刻理解是写出可靠程序的关键。比如，不能用（x-y<0）来取代（x<y），因为可能会产生溢出。甚至也不能用表达式（-y<-x）来取代，因为在二进制补码表示中负数和正数的范围是不对称的。算术溢出是程序错误的一个常见根源，然而很少有书从一个程序员的角度去讲述计算机算术的特性。

- 第 3 章：程序的机器级表示。我们教学生如何读由 C 编译器生成的 IA32 汇编语言。我们说明为不同控制结构，比如条件、循环和开关语句，生成的基本指令模式。我们还讲述过程的执行，包括栈分配、寄存器使用惯例和参数传递。我们讨论不同数据结构如结构、联合（union）和数组的分配和访问方式。学习本章的概念能够帮助学生成为更好的程序员，因为他们懂得他们的程序在机器上是如何表示的。另外一个妙处在于学生们对指针有了具体的了解。
- 第 4 章：处理器体系结构。这一章讲述基本的组合和时序逻辑元素，并展示这些元素在数据路径（datapath）中如何组合到一起，来执行 IA32 指令集的一个称为“Y86”的简化子集。我们从设计单时钟周期、非流水线化的数据路径开始，然后扩展成一个五阶段、流水线化的设计。本章中处理器设计的控制逻辑是用一种称为 HCL 的简单硬件描述语言来描述的。用 HCL 写的硬件设计能够编译和链接成本书中提供的图形处理器的模拟器。
- 第 5 章：优化程序性能。在这一章里，我们介绍许多提高代码性能的技术。我们从与机器无关的程序转换开始，这些标准是在任何机器上写任何程序时都应该遵循的。然后是那些功效有赖于目标机器和编译器特性的转换。为了促进这些转换，我们介绍了一个简单的操作模型，它描述了现代乱序（out-of-order）处理器是如何工作的，然后向学生们展示怎样利用这个模型来改进他们的 C 程序的性能。
- 第 6 章：存储器层次结构。对应用程序员来说，存储器系统是计算机系统中最直接可见的部分之一。到目前为止，学生们一直认同这样一个存储器系统概念模型，认为它是一个有一致访问时间的线性数组。实际上，存储器系统是一个由不同容量、造价和访问时间的存储设备组成的层次结构。我们讲述不同类型的随机存取存储器（RAM）和只读存储器（ROM）以及现代磁盘驱动器的几何形状和组织构造。我们描述这些存储设备是如何放置在层次结构中的，讲述访问局部性是如何使这种层次结构成为可能的。我们通过一个独特的观点使这些理论具体化、形象化，那就是将存储器系统视为“存储器山”，山脊是时间局部性，而斜坡是空间局部性。最后，我们向学生们阐述如何通过改善时间和空间局部性来提高应用程序的性能。
- 第 7 章：链接。本章讲述静态和动态链接，包括的概念有可重定位的（relocatable）和可执行的目标文件、符号解析、重定位（relocation）、静态库、共享目标库，以及与位置无关的代码。大多数系统书中都不涉及链接，而我们出于下面几个原因要讲述它。第一，学生们遇到的最迷惑的问题中，有一些是和链接时的小故障有关，尤其是对那些大型软件包来说。第二，链接器生成的目标文件是与一些像加载、虚拟存储器和存储器映射这样的概念相关的。
- 第 8 章：异常控制流。在课程的这个部分，我们通过介绍异常控制流（比如，正常分支和过程调用以外的控制流变化）的一般概念打破单一程序的模型。我们给出存在于系统所有层次的异常控制流的例子，从底层的硬件异常和中断，到并发进程的上下文切换，到 Unix 信号

本书的起源

本书起源于 1998 年秋季我们在卡内基梅隆（CMU）大学开设的一门编号为 15-213 的介绍性课程：计算机系统导论（Introduction to Computer System，ICS）[7]。从那以后，每学期都开设了 ICS 这门课程，每期有 150 名左右的学生，大多数是计算机科学和计算机工程专业二年级的学生。后来，这门课程还成为了卡内基梅隆大学计算机科学系以及电子和计算机工程系中大多数高级系统课程的先行必修课。

ICS 课程的宗旨是用一种不同的方式向学生介绍计算机。因为，我们的学生中几乎没有人有机会构造计算机系统。另一方面，大多数学生，甚至是计算机工程师，也要求日常能使用计算机和编写计算机程序。所以我们决定从程序员的角度来讲解系统，并采用这样的过滤方法：我们只讨论那些影响用户级 C 程序的性能、正确性或实用性的主题。

比如，我们排除了诸如硬件加法器和总线设计这样的主题。虽然我们谈及了机器语言，但是不关注如何编写汇编语言，而是关心 C 程序是如何被构造的，例如编译器是如何翻译指针、循环、过程调用和返回以及开关（switch）语句的。更进一步，我们将更广泛和现实地看待系统，包括硬件和系统软件，涵盖了链接、加载、进程、信号、性能优化、评估、I/O 以及网络与并发编程。

这种做法使得我们讲授 ICS 课程的方式对学生来讲既实用、具体，还能动手，同时也非常能调动学生的积极性。很快地，我们收到来自学生和教职工非常热烈和积极的反响，我们意识到卡内基梅隆大学以外的其他人也可以从我们的方法中获益。因此，历时两年，这本书从 ICS 课程笔记中应运而生了。

旁注：与 ICS 有关的数字

跟 ICS 课程有关的数字很特别。在第一学期过半的时候，我们发现课程的编号（15-213）正好就是卡内基梅隆大学的邮政编码，因此，还有了这样的话："15-213：给予卡内基梅隆大学精神的课程！"[1] 无独有偶，手稿的第一版是在 2001 年 2 月 13 日（2/13/01）印刷的。当我们在 SIGCSE[2] 教育会议上介绍这门课程时，被安排在了 213 房间，并且此书的最后一版有 13 个章节。好在我们并不迷信！

本书概述

本书由 13 章组成，旨在阐述计算机系统的核心概念。

- 第 1 章：计算机系统漫游。这一章通过研究"hello, world"这个简单程序的生命周期，介绍计算机系统的主要概念和主题。
- 第 2 章：信息的表示和处理。我们讨论计算机算术，重点描述对程序员有影响的无符号和二进制补码（two's complement）的数字表示法的特性。我们考虑数字是如何表示的，以及由此确定对于一个给定的字长，其可能编码值的范围。我们探讨有符号和无符号数字之间类型转换的效果，还阐述算术操作的数学特性。学生们很惊奇地了解到（二进制补码表示的）两个正数的和或者积可以为负。另一方面，二进制补码算法满足环的特性，因此，编译器可以

1 zip 既有"邮政编码"也有"精神"之意。——译者
2 SIGCSE 代表 Special Interest Group on Computer Science Education，计算机科学教育特殊兴趣组。——译者

传送引起的控制流突变，到 C 中破坏栈原则的非本地跳转（nonlocal jump）。

在这一章，我们还向学生们介绍进程的基本概念。学生们了解进程是如何工作的，以及如何在应用程序中创建和操纵进程。我们向他们展示应用程序员如何通过 Unix 系统调用使用多进程。学完本章，他们就能够编写带作业控制的 Unix 脚本了。

- 第 9 章：测量程序运行时间。这一章教给学生计算机是如何理解时间的［时间间隔计时器、CPU 周期计时器（cycle timer）和系统时钟］，当我们试图用这些时间来测量程序运行时时间的错误根源，以及怎样运用这些知识来得到准确的度量值。据我们所知，这是惟一的在以前还未以任何常规的方式讨论过的内容。我们在此讨论这个主题是因为它需要对汇编语言、进程和高速缓存有所了解。
- 第 10 章：虚拟存储器。我们讲述虚拟存储器系统是希望学生们对它的工作和特性有所了解。我们想让学生了解为什么不同的并发进程各自都有一个相同的地址范围，能共享某些页，但另外一些页又是独占的。我们还覆盖一些管理和操作虚拟存储器的问题。特别地，我们讨论了存储分配操作，比如 Unix 的 malloc 和 free 操作。阐述这些内容是出于下面几点目的。它加强了虚拟存储器空间只是字节数组，程序可以把它划分成不同存储单元的概念。它帮助学生理解包含有像存储泄漏和非法指针引用这样存储器引用错误的程序的后果。最后、许多应用程序员编写自己的优化了的存储分配操作来满足应用程序的需要和特性。
- 第 11 章：系统级 I/O。我们讲述 Unix I/O 的基本概念，例如文件和描述符。我们描述如何共享文件，I/O 重定向是如何工作的，还有如何访问文件的元数据。我们还开发了一个健壮的带缓冲区的 I/O 包，可以正确处理 short counts。我们阐述 C 的标准 I/O 库，以及它与 Unix I/O 的关系，重点谈到标准 I/O 的局限性，这些局限性使之不适合网络编程。总地说来，本章的论题是后面两章网络和并发编程的基础。
- 第 12 章：网络编程。对编程而言，网络是非常有趣的 I/O 设备，将许多我们前面文中学习的概念，比如进程、信号、字节顺序（byte ordering）、存储器映射和动态存储器分配，联系在一起。网络程序还为并发提供了强制性上下文，这是下一章的论题。本章是网络编程的细小片段，使学生们能够编写 Web 服务器。我们还讲述位于所有网络程序底层的客户端-服务器模型。我们展现了一个程序员对 Internet 的观点，并且教给学生们如何用套接字（socket）接口来编写 Internet 客户端和服务器。最后，我们介绍超文本传输协议 HTTP，并开发了一个简单的迭代式（iterative）Web 服务器。
- 第 13 章：并发编程。这一章以 Internet 服务器设计为例向学生们介绍了并发编程。我们比较对照了三种编写并发程序的基本机制（进程、I/O 多路复用技术以及线程），并且展示如何用它们来建造并发 Internet 服务器。我们探讨了用 P、V 信号操作、线程安全和可重入（reentrancy）、竞争条件以及死锁等来实现同步的基本原则。

可以基于本书的课程

指导教师可以使用本书来教授五种不同的系统课程（图 P.2）。特殊的课程则有赖于课程需要、个人品位、学生的背景和能力。图中的课程从左往右，逐渐强调以程序员的角度看待系统，以下是简单的描述：

- **ORG：** 一门以非传统风格介绍传统问题的计算机组成原理课程。传统的主题包括逻辑设计、处理器体系结构、汇编语言和存储器系统。然而，需要更多地强调对程序员的影响。例如，要反过来考虑数据表示对C程序的影响。学生们将学习到如何用机器语言来表示C结构。
- **ORG+：** ORG课程特别强调硬件对应用程序性能的影响。和ORG课程相比，学生要更多地学习代码优化和改进他们C程序的存储器性能。
- **ICS：** 基本的ICS课程，旨在培养开明的程序员，他们理解硬件、操作系统和编译系统对应用程序的性能和正确性的影响。和ORG+课程的一个显著不同是，本课程不论及低级处理器体系结构。相反地，程序员与现代乱序处理器的高级模型打交道。ICS课程非常适合安排成一个10周的学期，如果步调更从容一些，也可以延长为一个15周的学期。
- **ICS+：** 基本的ICS课程，额外论述一些系统编程问题，比如系统级I/O、网络编程和并发编程。这是一门一学期长度的卡内基梅隆大学课程，会讲述本书中除了低级处理器体系结构以外的每一章。
- **SP：** 一门系统编程课程。和ICS+课程相似，但是抛弃了浮点和性能优化，更加强调系统编程，包括进程控制、动态链接、系统级I/O、网络编程和并发编程。指导教师可能会想从其他渠道对某些高级论题做些补充，比如守护进程（daemon）、终端控制和Unix IPC（进程间通信）。

		课程				
章节	论题	ORG	ORG+	ICS	ICS+	SP
1	系统漫游	•	•	•	•	•
2	数据表示	•	•	•	•	⊙（d）
3	机器语言	•	•	•	•	•
4	处理器体系结构	•	•			
5	代码优化		•	•	•	
6	存储器层次结构	⊙（a）	•	•	•	⊙（a）
7	链接			⊙（c）	⊙（c）	•
8	异常控制流			•	•	•
9	性能测量				•	•
10	虚拟存储器	⊙（b）	•	•	•	•
11	系统级I/O				•	•
12	网络编程				•	•
13	并发编程				•	•

图P.2　五门基于本书的课程

注意：（a）只有硬件；（b）无动态存储分配；（c）无动态链接；（d）无浮点。ICS+是卡内基梅隆的15-213课程。

图P.2要表达的主要信息是本书给了你多种选择。如果你希望你的学生更多地了解低级处理器体系结构，那么通过ORG和ORG+课程可以达到目的。另一方面，如果你想将当前的计算机组成课程转换成ICS或者ICS+课程，但是又担心突然做这样猛烈的变化，那么你可以逐步递增转向ICS课程。你可以从OGR课程开始，它以一种非传统的方式教授传统的问题。一旦你对这些内容感到驾轻就熟

了，就可以转到ORG+，最终转到ICS。如果学生没有C的经验（比如他们只用Java编过程序），你可以花几周的时间在C上，然后再讲述ORG或者ICS课程的内容。

最后，我们认为ORG+和SP课程适合安排为两期（两个季度或者两个学期）。或者你可以考虑按照一期ICS和一期SP的方式来教授ICS+课程。

课堂测试的实验练习

卡内基梅隆大学的ICS+课程得到了学生们很高的评价。这门课的中值分数一般为5.0/5.0，平均分数一般为4.6/5.0。学生们表扬说这门课非常有趣，令人兴奋，主要就是因为相关的实验练习。下面是本书提供的一些实验的示例。

- 数据实验。这个实验要求学生们实现简单的逻辑和算术函数，但是只能使用一个高度受限的C的子集。比如，他们必须只能用位级操作来计算一个数字的绝对值。这个实验帮助学生们了解C数据类型的位级表示，和数据操作的位级行为。
- 二进制炸弹实验。二进制炸弹是一个作为目标代码文件提供给学生们的程序。运行时，它提示用户输入6个不同的字符串。如果其中的任何一个不正确，炸弹就会“爆炸”，打印出一条错误信息，并且在分级（grading）服务器上记录事件日志。学生们必须通过对程序反汇编和逆向工程来测定应该是哪6个串，从而解除他们各自炸弹的雷管。该实验教会学生理解汇编语言，并且强制他们学习怎样使用调试器。
- 缓冲区溢出实验。它要求学生们通过研究一个缓冲区溢出的错误，来修改二进制可执行文件的运行时行为。这个实验教会学生们栈的原理，并让他们了解到写那种易于遭受缓冲区溢出攻击的代码的危险性。
- 体系结构实验。第4章的几个家庭作业问题能够组合成一个实验作业，在实验中，学生们修改处理器的HCL描述以增加新的指令、修改分支预测策略，或者增加或删除旁路路径和寄存器端口。设计出来的处理器能够被模拟，并通过运行自动化测试检测出大多数可能的错误。这个实验使学生们能体验到处理器设计中令人激动的部分，而不需要他们学习和建造用Verilog或者VHDL语言写的复杂而低级的模块。
- 性能实验。学生们必须优化应用的核心函数（比如卷积积分或矩阵转置）的性能。这个实验非常清晰地表明了高速缓存的特性，并给学生们低级程序优化的经验。
- shell实验。学生们实现他们自己的带有作业控制的Unix shell程序，包括ctrl-c和ctrl-z按键、fg、bg和jobs命令。这是学生们第一次接触并发，并且让他们对Unix的进程控制、信号和信号处理有清晰的了解。
- malloc实验。学生们实现他们自己的malloc、free和realloc（可选地）版本。这个实验让学生们清晰地理解数据的布局和组织，并且要求他们评估时间和空间效率的各种权衡和折中。
- 代理实验。学生们实现一个位于浏览器和万维网其他部分之间的并行Web代理。这个实验向学生们揭示了Web客户端和服务器这样的问题，并且联系起了课程中许多概念，比如字节排序、文件I/O、进程控制、信号、信号处理、存储器映射、套接字和并发。

本书的教师手册有对实验的详细讨论，还有关于下载支持软件的说明。

致　谢

我们衷心地感谢那些给了我们中肯批评和鼓励的众多朋友及同事。特别感谢我们 15-213 课程的学生们，他们充满感染力的精力和热情鞭策我们前行。Nick Carter 和 Vinny Furia 无私地提供了他们的 malloc 程序包。

Guy Blelloch、Greg Kesden、Bruce Maggs 和 Todd Mowry 在多个学期中教授此课，给我们鼓励并帮助改进课程内容。Herb Derby 提供了早期的精神指导和鼓励。Allan Fisher、Garth Gibson、Thomas Gross、Satya、Peter Steenkiste 和 Hui Zhang 从一开始就鼓励我们开设这门课程。Garth 早期给的建议促使本书的工作得以开展，并且在 Allan Fisher 领导的小组的帮助下又细化和修订了本书的工作。Mark Stehlik 和 Peter Lee 提供了极大的支持，使得这些内容成为本科生课程的一部分。Greg Kesden 针对 ICS 在操作系统课程上的影响提供了有益的反馈意见。Greg Ganger 和 Jiri Schindler 提供了一些磁盘驱动的描述说明，并回答了我们关于现代磁盘的疑问。Tom Striker 向我们展示了存储器山的比喻。James Hoe 在处理器体系结构方面提出了很多有用的建议和反馈。

有一群特殊的学生极大地帮助我们发展了这门课程的内容，他们是 Khalil Amiri、Angela Demke Brown、Chris Colohan、Jason Crawford、Peter Dinda、Julio Lopez、Bruce Lowekamp、Jeff Pierce、Sanjay Rao、Balaji Sarpeshkar、Blake Scholl、Sanjit Seshia、Greg Steffan、Tiankai Tu、Kip Walker 和 Yinglian Xie。尤其是 Chris Colohan 建立了愉悦的氛围并持续到今天，还发明了传奇般的“二进制炸弹”，这是一个对教授机器语言代码和调试概念非常有用的工具。

Chris Bauer、Alan Cox、Peter Dinda、Sandhya Dwarkadis、John Greiner、Bruce Jacob、Barry Johnson、Don Heller、Bruce Lowekamp、Greg Morrisett、Brian Noble、Bobbie Othmer、Bill Pugh、Michael Scott、Mark Smotherman、Greg Steffan 和 Bob Wier 花费了大量时间阅读此书的早期草稿，并给予我们建议。特别感谢 Peter Dinda（西北大学）、John Greiner（莱茨大学）、Wei Hsu（明尼苏达大学）、Bruce Lowekamp（威廉&玛丽大学）、Bobbie Othmer（明尼苏达大学）、Michael Scott（罗彻斯特大学）和 Bob Wier（落基山学院）在教学中测试此书的试用版。同样特别感谢他们的学生们！

我们还要感谢 Prentice Hall 出版公司的同事。感谢 Marcia Horton、Eric Frank 和 Harold Stone 不懈地支持和远见。Harold 还帮我们提供了对 RISC 和 CISC 处理器体系结构准确的历史观点。Jerry Ralya 有惊人的见识，并教会了我们很多如何写作的知识。

最后，我们衷心感谢伟大的技术作家 Brian Kernighan 以及后来的 W. Richard Stevens，他们向我们证明了技术书籍也能写得如此优美。

谢谢你们所有的人。

Randy Bryant
Dave O'Hallaron

关于作者

Randal E. Bryant 1973年获得密歇根大学（University of Michigan）学士学位，随即就读麻省理工学院（Massachusetts Institute of Technology）的研究生院，并在1981年获计算机博士学位。他在加州理工学院（California Institute of Technology）做了三年助教，从1984年至今一直是卡内基梅隆大学（Carnegie Mellon）的教师。他现在是计算机科学的主任教授（president's professor）和计算机科学系的系主任。他同时还受邀于电子和计算机工程系。

他从事本科和研究生计算机系统方面课程的教学超过20年。在讲授计算机体系结构课程多年后，他开始把关注点从如何设计计算机转移到程序员如何在更好的了解系统的情况下编写出更有效和更可靠的程序。他和O'Hallaron教授一起在卡内基梅隆大学开设了“计算机系统导论”课程，那便是此书的基础。他还教授一些算法和编程方面的课程。

Bryant教授的研究涉及帮助硬件设计者验证其系统正确性的软件工具的设计。其中，包括几种类型的模拟器，以及用数学方法来证明设计正确性的形式化验证工具。他发表了100多篇技术论文。包括Intel、Motorola、IBM和Fujitsu在内的主要计算机制造商都使用他的研究成果。他还因他的研究获得过数项大奖。其中包括Semiconductor Research Corporation颁发的两个发明荣誉奖和一个技术成就奖，ACM（Association for Computer Machinery，美国计算机学会）颁发的Kanellakis理论与实践奖，还有IEEE（Institute of Electrical and Electronics Engineers，电气和电子工程师协会）授予的W. R. G Baker奖和50年金质奖章（a Golden Jubilee Medal）。他同时是ACM和IEEE的院士。

David R. O'Hallaron 1986年在维吉尼亚大学（University of Virginia）获得计算机科学的博士学位。在通用电气工作一段时间后，于1989年作为系统科学家成为卡内基梅隆大学的教员。他目前是计算机科学系和电子及计算机工程系的副教授。

他教授一些本科生和研究生的计算机系统方面的课程，例如计算机体系结构、计算机系统绪论、并行处理器设计和Internet服务。和Bryant教授一起，他开设了“计算机系统导论”课程，那便是此书的基础。

O'Hallaron教授和他的学生从事计算机系统领域的研究。特别地，他们开发了一些软件系统，帮助科学家和工程师在计算机上模拟自然界。其中最著名的是Quake项目，一群计算机科学家、土木工程师和地震学家致力发展预测在强烈地震中大地运动的能力，这些强烈地震包括南加洲、古巴、日本、墨西哥和新西兰的大地震。同Quake项目中其他人员一起，他获得了CMU计算机科学学院颁发的Allen Newell优秀研究奖章。他为Quake项目创立的基准程序183.equake，被SPEC[1]选入非常有影响力的SPEC CPU和OMP（Open Mp）基准程序包中。

---

1 SPEC是Standards Performance Evaluation Corporation的缩写。——译者

目 录

第2部分 在系统上运行程序

第 3 部分 程序间的交互和通信

CHAPTER 1

计算机系统漫游

计算机系统是由硬件和系统软件组成的，它们共同工作来运行应用程序。虽然系统的具体实现方式随着时间不断变化，但是系统内在的概念却没有改变。所有计算机系统都由相似的硬件和软件组成，它们又执行着相似的功能。这本书是为这样一些程序员而写的，他们希望通过了解这些部件如何工作以及如何影响程序的准确性和性能，来提高自身技能。

你现在就要开始一次有趣的漫游历程了。如果你全力投身学习本书中的概念，理解底层计算机系统的本质和它如何影响你的应用程序，那么它将指引你步上成为稀缺的“权威程序员”的道路。

你将开始学习一些实践技巧，比如如何避免由计算机表示数字方式引起的奇怪的数字错误。你将学会怎样通过一些聪明的小窍门来优化你的 C 代码，这些小窍门运用了现代处理器和存储器（memory）系统的设计。你将了解到编译器是如何实现过程调用的，并且了解到如何利用这个知识来避免缓存区溢出错误带来的安全漏洞，这些错误给网络和 Internet 软件带来了巨大的麻烦。你将学会如何认识和避免链接时那些令人讨厌的错误，它们困扰着普通的程序员。你将学会如何编写自己的 Unix shell、自己的动态存储分配包，甚至于自己的 Web 服务器！

在 Kernighan 和 Ritchie 的关于 C 编程语言的经典文章[40]中，他们通过图 1.1 中所示的 hello 程序来向读者介绍 C。尽管 hello 程序是一个非常简单的程序，但是为了完成它的执行，系统的每个主要组成部分都需要协调工作。从某种意义上来说，本书的目的就是要帮助你了解当你在系统上执行 hello 程序时，系统发生了什么以及为什么会如此运作。

code/intro/hello.c

```
1   #include <stdio.h>
2
3   int main()
4   {
5         printf("hello, world\n");
6   }
```

code/intro/hello.c

图 1.1 hello 程序

我们通过跟踪 hello 程序的生命周期来开始我们对系统知识的学习，它的生命周期从它被程序员创建开始，包括在系统上运行、输出简单的消息，然后终止。我们将沿着这个程序的生命周期，简要地介绍一些逐步出现的关键概念、专业术语和成分。后面的章节将围绕这些内容展开。

1.1 信息就是位+上下文

我们的 hello 程序的生命是从一个源程序（或者说源文件）开始的，该源程序由程序员通过编辑器创建并保存为文本文件，文件名就是 hello.c。源程序实际上就是一个由 0 和 1 组成的位（又称为比特）序列，这些位被组织成 8 个一组，称为字节。每个字节都表示程序中某个文本字符。

大部分的现代系统都使用 ASCII 标准来表示文本字符，这种方式实际上就是用一个惟一的字节大小的整数值来表示每个字符。比如，图 1.2 中给出了 hello.c 程序的 ASCII 表示。

hello.c 程序是以字节序列的方式储存在文件中的。每个字节都有一个整数值，对应于某个字符，例如，第一个字节的整数值是 35，它对应的就是字符“#”。第二个字节整数值为 105，它对应的字符

是“i”，以此类推。注意，每行文本都是以一个看不见的换行符“\n”来结束的，它所对应的整数值为 10。像 hello.c 这样只由 ASCII 字符构成的文件称为文本文件，所有其他文件则称为二进制文件。

```
  #    i    n    c    l    u    d    e  <sp>   <    s    t    d    i    o    .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46
  h    >   \n   \n    i    n    t  <sp>   m    a    i    n    (    )   \n    {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123
 \n <sp> <sp> <sp> <sp>   p    r    i    n    t    f    (    "    h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108
  l    o    ,  <sp>   w    o    r    l    d    \    n    "    )    ;   \n    }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

图 1.2 hello.c 的 ASCII 文本表示

hello.c 的表示方法说明了一个基本的思想：系统中所有的信息——包括磁盘文件、存储器中的程序、存储器中存放的用户数据以及网络上传送的数据，都是由一串比特表示的。区分不同数据对象的惟一方法是我们读到这些数据对象时的上下文。比如，在不同的上下文中，同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令。

作为程序员，我们需要了解数字的机器表示方式，因为它们与常见的整数和实数是不同的。它们有些相似，但这种相似并不为人所知。这方面的基本原理将在第 2 章中详细描述。

旁注：C 编程语言

C 语言是贝尔实验室的 Dennis Ritchie 于 1969 年～1973 年间创建的。美国国家标准化组织（American National Standards Institute，ANSI）在 1989 年颁布了 ANSI C 的标准，该标准定义了 C 语言和一系列函数库，即所谓的 C 标准库。Kernighan 和 Ritchie 在他们众所周知的经典著作“K&R”[40]中描述了 ANSI C。用 Ritchie 的话来说，C 是“古怪的、有缺陷的，但同时也是一个巨大的成功”。为什么说是成功的呢？

- C 与 Unix 操作系统关系密切。C 从开始就是作为一种用于 Unix 系统的程序语言开发出来的。Unix 内核的大部分，以及所有它支持的工具和函数库都是用 C 语言编写的。20 世纪 70 年代后期到 80 年代早期，Unix 风行于高等院校，许多人开始接触 C 并喜欢上了 C。因为 Unix 几乎全部是用 C 编写的，它就可以很方便地移植到新的机器上，这种特点为 C 和 Unix 赢来了更为广泛的支持。
- C 是一个小而简单的语言。C 语言的设计是由一个人而非一个协会掌控的，其结果就是这是一个简洁明了、没有什么冗赘的设计。K&R 这本书用了大量的例子和练习描述了完整的 C 语言及其标准库，而全书不过 261 页。C 语言的简单使它相对而言易于学习，也易于移植到不同的计算机上。
- C 是为实践目的设计的。C 是设计用来实现 Unix 操作系统的。后来，其他人发现能够用这门语言无障碍地编写他们想要的程序。

C 语言是系统级编程的首选，同时它也非常适用于应用级程序的编写。然而，它也并非适用于所有的程序员和所有的情况。C 的指针是造成困惑和程序错误的一个常见原因。同时，C 还缺乏对一些有用抽象的显式支持，例如类、对象和异常。针对应用级程序的 C++和 Java 等新的程序语言解决了这些问题。

1.2 程序被其他程序翻译成不同的格式

在hello程序生命周期的一开始时是一个高级C程序，因为当处于这种形式时，它是能够被人读懂的。然而，为了在系统上运行hello.c程序，每条C语句都必须被其他程序转化为一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序（executable object program）的格式打好包，并以二进制磁盘文件的形式存放起来。目标程序也称为可执行目标文件（executable object file）。

在Unix系统上，从源文件到目标文件的转化是由编译器驱动程序（compiler driver）完成的：

```
unix> gcc -o hello hello.c
```

在这里，gcc编译器驱动程序读取源程序文件hello.c，并把它翻译成一个可执行目标文件hello。这个翻译的过程是分为四个阶段完成的，如图1.3所示。执行这四个阶段的程序（预处理器、编译器、汇编器和链接器）一起构成了编译系统。

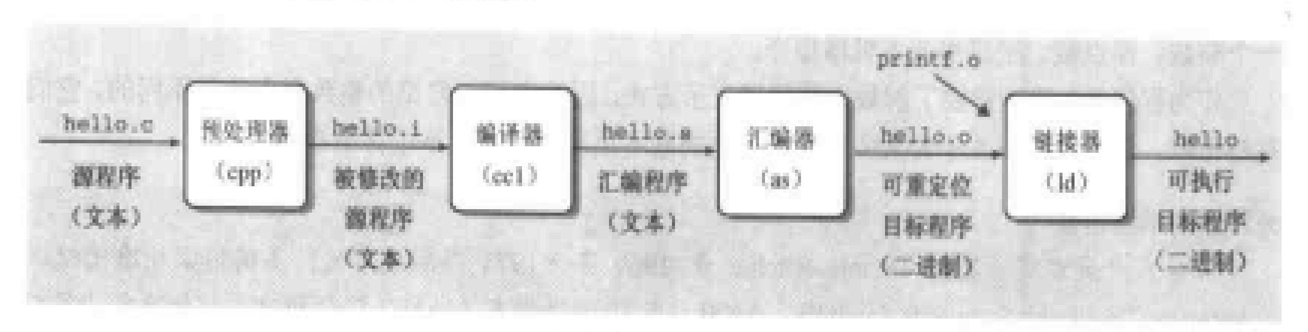


图1.3 编译系统

- 预处理阶段。预处理器（cpp）根据以字符#开头的命令（directives），修改原始的C程序。比如hello.c中第一行的#include <stdio.h>指令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入到程序文本中去。结果就得到了另一个C程序，通常是以.i作为文件扩展名。
- 编译阶段。编译器（cc1）将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切地描述了一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言。例如，C编译器和Fortran编译器产生的输出文件用的都是一样的汇编语言。
- 汇编阶段。接下来，汇编器（as）将hello.s翻译成机器语言指令，把这些指令打包成为一种叫做可重定位（relocatable）目标程序的格式，并将结果保存在目标文件hello.o中。hello.o文件是一个二进制文件，它的字节编码是机器语言指令而不是字符。如果我们在文本编辑器中打开hello.o文件，呈现的将是一堆乱码。
- 链接阶段。请注意，我们的hello程序调用了printf函数，它是标准C库中的一个函数，每个C编译器都提供。printf函数存在于一个名为printf.o的单独的预编译目标文件中，而这个文件必须以某种方式并入到我们的hello.o程序中。链接器（ld）就负责处理这种并入，结果就得到hello文件，它是一个可执行目标文件（或者简称为可执行文件）。可执行文件加载到存储器后，由系统负责执行。

旁注：GNU 项目

GCC 是 GNU（GNU 是 GNU's Not Unix 的简写）项目开发出来的众多有用工具之一。GNU 项目是 1984 年由 Richard Stallman 发起的一个免税的慈善项目。该项目的目标非常宏大，就是开发出一个完整的类 Unix 的系统，其源代码能够不受限制地被修改和传播。到 2002 年，GNU 项目已经发展成为一个 Unix 操作系统的所有主要部件构成的环境，但内核除外，内核是由 Linux 项目独立发展而来的。GNU 环境包括 EMACS 编辑器、GCC 编译器、GDB 调试器、汇编器、链接器、处理二进制文件的工具以及其他一些部件。

GNU 项目取得了非凡的成绩，但是却常常被忽略。现代开放源码运动（通常和 Linux 联系在一起）的思想起源是 GNU 项目中自由软件（free software）的概念。[此处的 free 为自由言论（free speech）中"自由"之意，而非免费啤酒（free beer）中"免费"之意。] 而且，Linux 的知名度在很大程度上还要归功于 GNU 工具，它们给 Linux 内核提供了环境。

1.3 了解编译系统如何工作是大有益处的

对于像 hello.c 这样简单的程序，我们可以依靠编译系统生成正确有效的机器代码。但是，有一些重要的原因促使程序员必须知道编译系统是如何工作的。

- **优化程序性能**。现代编译器都是成熟的工具，通常可以生成很好的代码。作为程序员，我们无须为了写出高效代码而去了解编译器的内部工作。但是，为了在我们的 C 程序中做出好的代码选择，我们确实需要对汇编语言以及编译器如何将不同的 C 语句转化为汇编语言有一些基本的了解。比如，一个 switch 语句是不是总是比一系列的 if-then-else 语句高效得多？一个函数调用的代价有多大？while 循环比 do 循环更有效吗？指针引用比数组索引更有效吗？相对于用通过引用传递过来的参数求和，为什么用本地变量求和的循环，其运行就会快得多呢？为什么两个功能相近的循环的运行时间会有很大差异？

  在第 3 章中，我们将介绍 Intel IA32 机器语言，并阐述编译器是如何将不同的 C 程序结构翻译成机器语言的。在第 5 章中，你将学习如何通过对 C 代码做些简单转换，帮助编译器更好地完成工作，从而调整你的 C 程序的性能。然后在第 6 章，你将学习存储器系统的层次特性，C 编译器是如何将数组存放在存储器中，以及你的 C 程序又是如何能够利用这些知识从而更高效地运行。
- **理解链接时出现的错误**。根据我们的经验，一些最令人困扰的程序错误往往都与链接器操作有关，尤其是当你试图建立大型的软件系统时。比如，链接器报告说它无法解析一个引用，这是什么意思？静态变量和全局变量的区别是什么？如果你在不同的 C 文件中定义了名字相同的两个全局变量会发生什么？静态库和动态库的区别是什么？为什么我们在命令行上排列库的顺序是有影响的？最为烦人的是，为什么有些链接错误直到运行时才出现？在第 7 章中，你将了解到这些问题的答案。
- **避免安全漏洞**。近年来，缓冲区溢出错误造成了大多数网络和 Internet 服务器上的安全漏洞。这些错误的存在是因为太多的程序员忽视了编译器用来为函数产生代码的堆栈规则。作为学习汇编语言的一部分，我们将在第 3 章中描述堆栈规则和缓冲区溢出错误。

1.4 处理器读并解释储存在存储器中的指令

此刻，我们的 hello.c 源程序已经被编译系统转换成了可执行目标文件 hello，并被存放在磁盘上。为了在 Unix 系统上运行该可执行文件，我们将它的文件名输入到称为 shell 的应用程序中：

```
unix> ./hello
hello, world
unix>
```

shell 是一种命令行解释器，它输出一个提示符，等待你输入一行命令，然后执行这个命令。如果该命令行的第一个单词不是一个内置的 shell 命令，那么 shell 就会假设这是一个可执行文件的名字，要加载和执行该文件。所以在此例中，shell 将加载和执行 hello 程序，然后等待程序终止。hello 程序在屏幕上输出它的信息，然后终止。shell 随后输出一个提示符，等待下一个输入的命令行。

1.4.1 系统的硬件组成

为了了解运行时 hello 程序发生了什么，我们需要理解一个典型系统的硬件组织，如图 1.4 所示。这张图是 Intel Pentium 系统产品族的模型，但是所有其他系统也有相同的外观和特性。现在不要担心这张图很复杂——我们将在贯穿这本书的课程中分阶段介绍大量的细节。

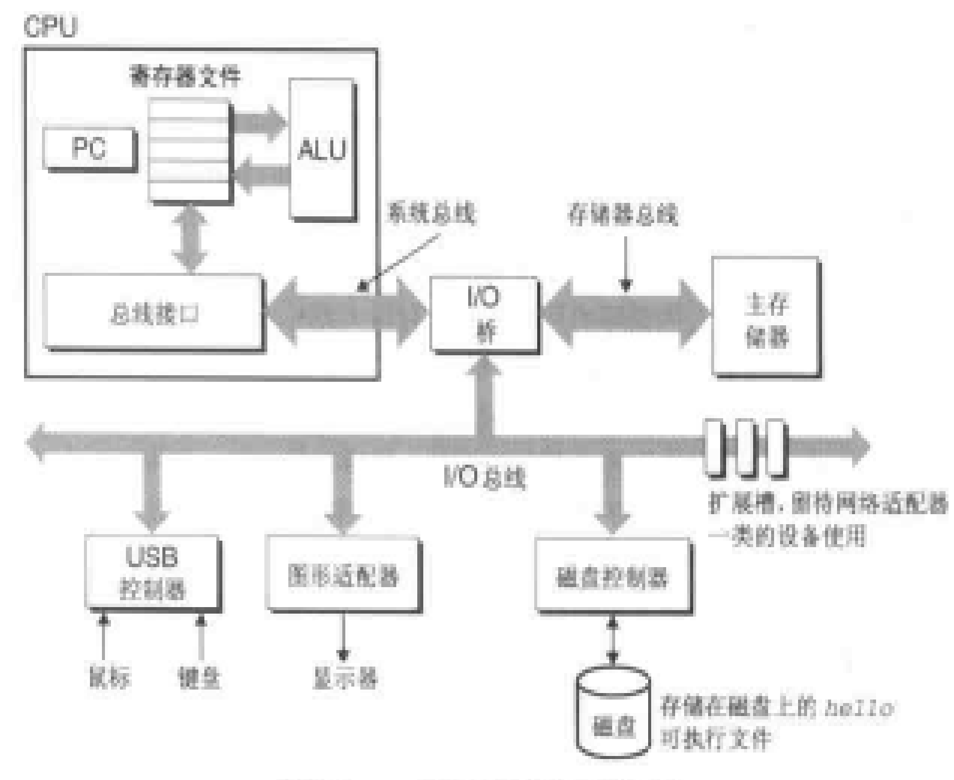


图 1.4 一个典型系统的硬件组成

CPU：中央处理单元；ALU：算术/逻辑单元；PC：程序计数器；USB：通用串行总线。

总线

贯穿整个系统的是一组电子管道，称做总线，它携带信息字节并负责在各个部件间传递。通常总线被设计成传送定长的字节块，也就是字（word）。字中的字节数（即字长）是一个基本的系统参数，各

个系统中也不尽相同。比如，Intel Pentium 系统的字长为 4 字节，而服务器类的系统，例如 Intel Itaniums 和高端的 Sun 公司的 SPARCS 的字长为 8 字节。用于汽车和工业中的嵌入式控制器之类较小的系统的字长往往只有 1 或 2 字节。为了便于描述，我们假设字长为 4 字节，并且假设总线一次只传 1 个字。

I/O 设备

I/O（输入/输出）设备是系统与外界的联系通道。我们的示例系统包括四个 I/O 设备：作为用户输入的键盘和鼠标，作为用户输出的显示器，以及用于长期存储数据和程序的磁盘驱动器（简单地说就是磁盘）。最开始，可执行程序 hello 就放在磁盘上。

每个 I/O 设备都是通过一个*控制器*或*适配器*与 I/O 总线连接起来的。控制器和适配器之间的区别主要在于它们的组成方式。控制器是 I/O 设备本身中或是系统的主印制电路板（通常被称做主板）上的芯片组，而适配器则是一块插在主板插槽上的卡。无论如何，它们的功能都是在 I/O 总线和 I/O 设备之间传递信息。

第 6 章会更多地说明磁盘之类的 I/O 设备是如何工作的。在第 11 章中，你将学习如何在应用程序中利用 Unix I/O 接口访问设备。我们尤其关注特别有趣的网络类设备，不过这些技术也适用于其他设备。

主存

主存是一个临时存储设备，在处理器执行程序时，它被用来存放程序和程序处理的数据。物理上来说，主存是由一组 DRAM（动态随机存取存储器）芯片组成的。逻辑上来说，存储器是由一个线性的字节数组组成的，每个字节都有自己惟一的地址（数组索引），这些地址是从零开始的。一般来说，组成程序的每条机器指令都由不定量的字节构成。与 C 程序变量相对应的数据项的大小是根据类型变化的。比如，在运行 Linux 的 Intel 机器上，short 类型的数据需要 2 字节，int、float 和 long 类型则需要 4 字节，而 double 类型需要 8 字节。

第 6 章具体说明存储技术，比如 DRAM 是如何工作的，以及它们又是如何组合起来构成主存的。

处理器

中央处理单元（CPU）简称处理器，是解释（或*执行*）存储在主存中指令的引擎。处理器的核心是一个被称为*程序计数器*（PC）的字长大小的存储设备（或*寄存器*）。在任何一个时间点上，PC 都指向主存中的某条机器语言指令（内含其地址）。[1]

从系统通电开始，直到系统断电，处理器一直在不假思索地重复执行相同的基本任务：从程序计数器（PC）指向的存储器处读取指令，解释指令中的位，执行指令指示的简单操作，然后更新程序计数器指向下一条指令，而这条指令并不一定在存储器中和刚刚执行的指令相邻。

这样的简单操作的数目并不多，它们在主存、寄存器文件（register file）和*算术逻辑单元*（ALU）之间循环。寄存器文件是一个小的存储设备，由一些字长大小的寄存器组成，这些寄存器每个都有惟一的名字。ALU 计算新的数据和地址值。下面是一些简单操作的例子，CPU 在指令的要求下可能会执行这些操作。

- *加载*：从主存拷贝一个字节或者一个字到寄存器，覆盖寄存器原来的内容。
- *存储*：从寄存器拷贝一个字节或者一个字到主存的某个位置，覆盖这个位置上原来的内容。

1 PC 也普遍地被用来作为个人计算机的缩写。然而，两者之间的区别应该可以很清楚地从上下文中看出来。

- **更新**：拷贝两个寄存器的内容到ALU，ALU将两个字相加，并将结果存放到一个寄存器中，覆盖该寄存器中原来的内容。
- **I/O读**：从一个I/O设备中拷贝一个字节或者一个字到一个寄存器。
- **I/O写**：从一个寄存器中拷贝一个字节或者一个字到一个I/O设备。
- **转移**：从指令本身中抽取一个字，并将这个字拷贝到程序计数器（PC）中，覆盖PC中原来的值。

第4章将对处理器的工作原理给予更详细的说明。

1.4.2 执行hello程序

通过对系统的硬件组成和操作的简单学习，我们开始能够了解当我们运行示例程序时发生了什么。在这里我们必须忽略很多细节，稍后会做一些补充，但是现在我们将很满意于这种粗略的描述。

首先，shell程序执行它的指令，等待我们输入命令。当我们在键盘上输入字符串“./hello”后，shell程序就逐一读取字符到寄存器，再把它存放到存储器中，如图1.5所示。

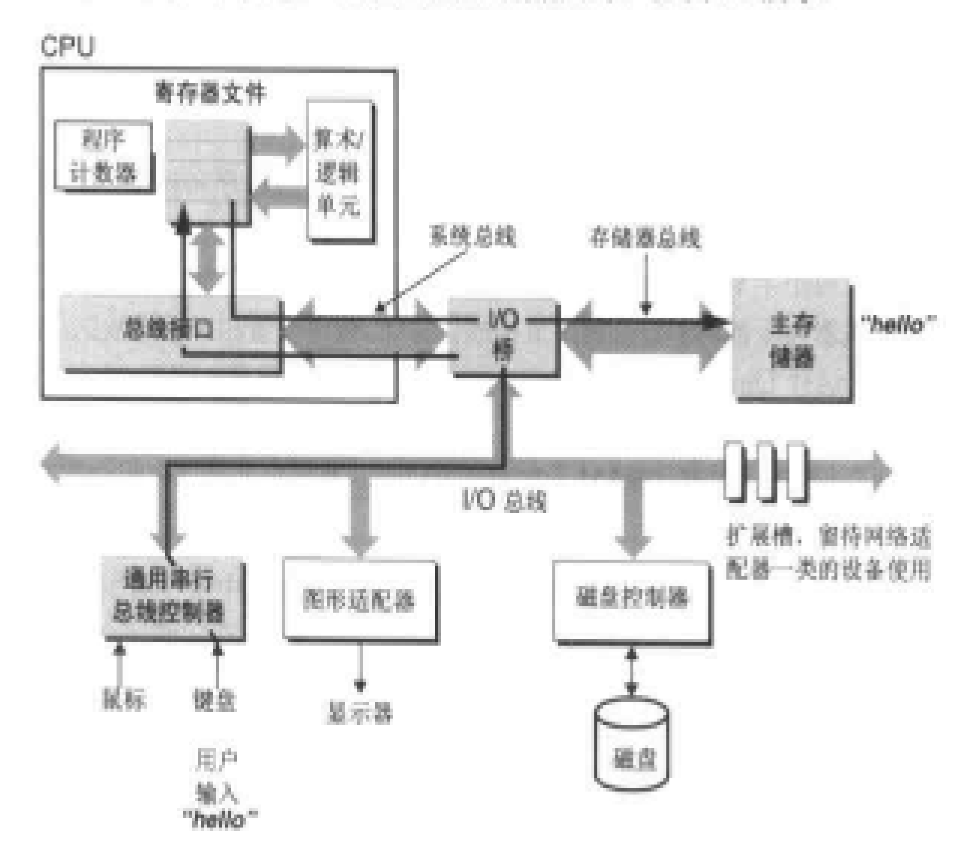


图1.5 从键盘上读取hello命令

当我们在键盘上敲回车键时，shell就知道我们已经结束了命令的输入。然后shell执行一系列指令，这些指令将hello目标文件中的代码和数据从磁盘拷贝到主存，从而加载hello文件。数据包括最终会被输出的字符串“hello, world\n”。

利用称为DMA（直接存储器存取，将在第6章中讨论）的技术，数据可以不通过处理器而直接从磁盘到达主存。这个步骤如图1.6所示。

一旦hello目标文件中的代码和数据被加载到了存储器，处理器就开始执行hello程序的主程序中

的机器语言指令。这些指令将“hello, world\n”串中的字节从存储器中拷贝到寄存器文件，再从寄存器中文件拷贝到显示设备，最终显示在屏幕上。这个步骤如图 1.7 所示。

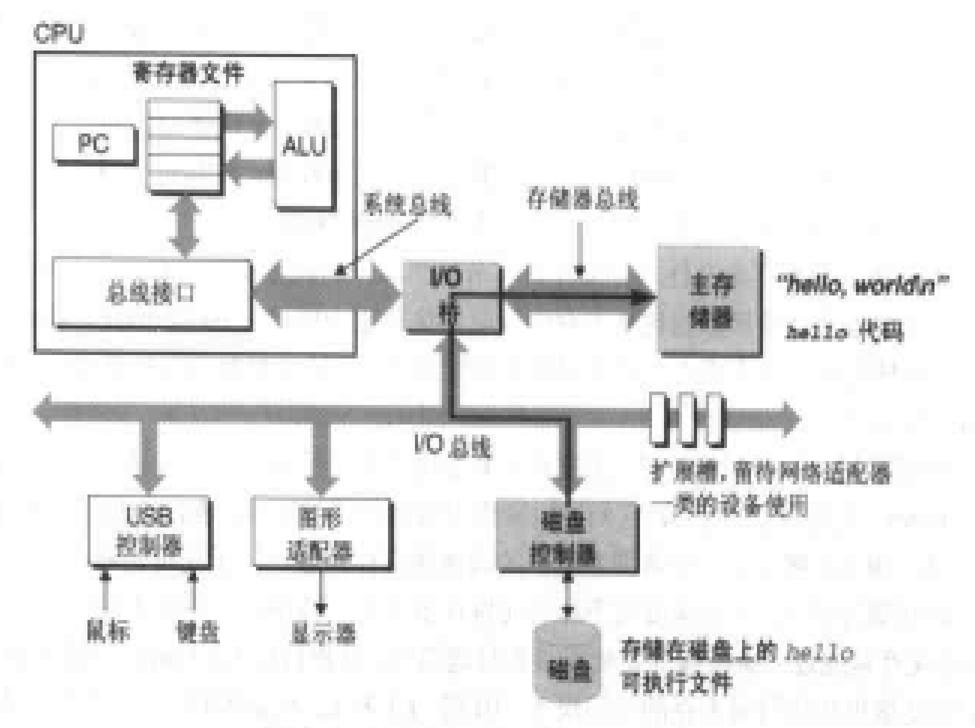


图 1.6 从磁盘加载可执行文件到主存

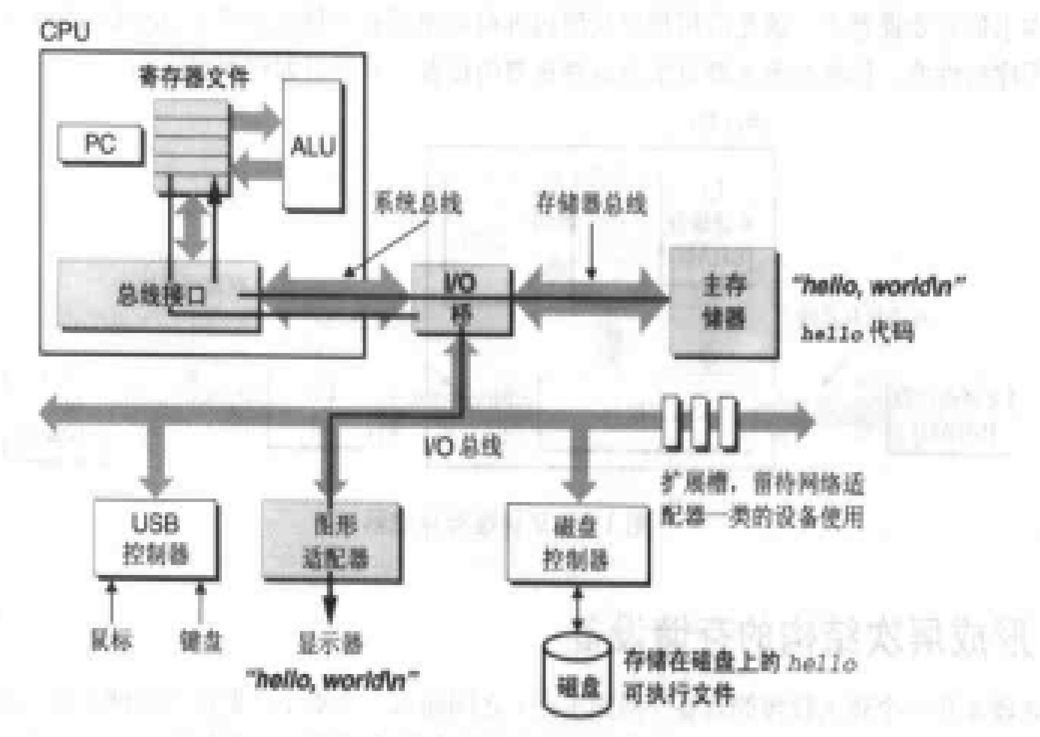


图 1.7 从存储器写输出串到显示器

1.5 高速缓存

通过这个简单的示例我们了解到重要的一课，那就是系统花费了大量的时间把信息从一个地方挪

到另一个地方。hello 程序的机器指令最初是存放在磁盘上的，当程序加载时，它们被拷贝到主存。当处理器运行程序时，指令又从主存拷贝到处理器。相似地，数据串“hello, world/n”开始时在磁盘上，再被拷贝到主存，然后从主存上拷贝到显示设备。从一个程序员的角度来看，大量的拷贝减慢了程序的实际工作。因此，系统设计者的一个主要目标就是使这些拷贝操作尽可能的快。

根据机械原理，较大的存储设备要比较小的存储设备运行得慢，而快速设备的造价远高于低速同类设备。比如说，一个典型系统上的磁盘驱动器可能比主存大 100 倍，但是对处理器而言，从磁盘驱动器上读取一个字的时间开销要比从主存中读取的开销大 1000 万倍。

类似地，一个典型的寄存器文件只存储几百字节的信息，与此相反，主存里可存放几百万字节。然而，处理器从寄存器文件中读数据比从主存中读取要快几乎 100 倍。更麻烦的是，随着这些年半导体技术的进步，这种处理器与主存之间的差距还在持续增大。加快处理器的运行速度比加快主存的处理速度要容易和便宜得多。

针对这种处理器与主存之间的差异，系统设计者采用了更小更快的存储设备，称为高速缓存存储器（cache memories，简称高速缓存），它们被用来作为暂时的集结区域，存放处理器在不久的将来可能会需要的信息。图 1.8 展示了一个典型系统中的高速缓存存储器。位于处理器芯片上的 L1 高速缓存的容量可以达到数万字节，访问速度几乎和访问寄存器文件一样快。一个容量为数十万到数百万的更大的 L2 高速缓存是通过一条特殊的总线连接到处理器的。进程访问 L2 的时间开销要比访问 L1 的开销大 5 倍，但是这仍然比访问主存的时间快 5~10 倍。L1 和 L2 高速缓存是用一种叫做静态随机访问存储器（SRAM）的硬件技术实现的。

这本书的重要课题之一就是应用程序员通过理解高速缓存存储器的机理，能够利用这些知识极大地提高程序的性能。你将在第 6 章里学习这些重要的设备，并学习如何利用它们。

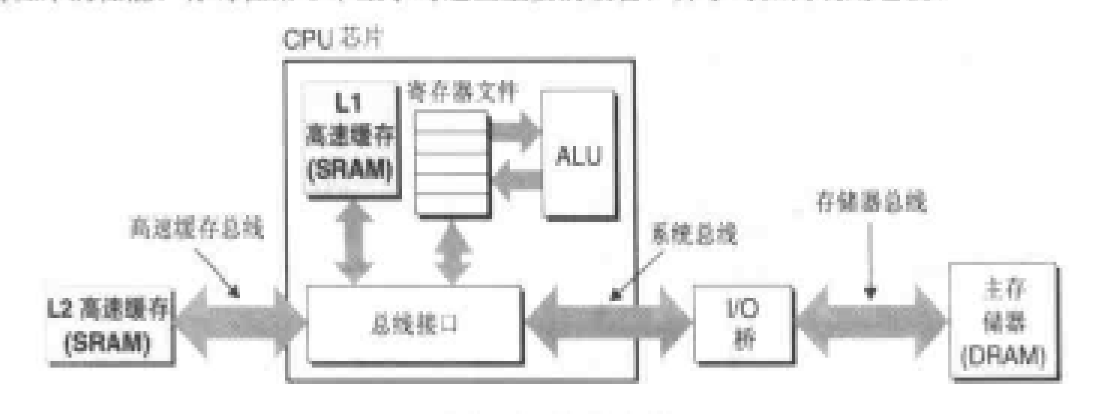


图 1.8 高速缓存存储器

1.6 形成层次结构的存储设备

在处理器和一个较大较慢的设备（例如主存）之间插入一个更小、更快的存储设备（例如，高速缓存存储器）的想法成为一个普遍的观念。实际上，每个计算机系统中的存储设备都被组织成一个存储器层次模型，就像图 1.9 所展示的那样。在这个层次模型中，从上至下，设备变得更慢、更大，并且每字节的造价也更便宜。寄存器文件在层次模型中位于最顶部，也就是第 0 级或记为 L0。L1 高速缓存处在第一层（所以称为 L1），L2 高速缓存占据第二层，主存在第三层，以此类推。

存储器分层结构的主要思想是一个层次上的存储器作为下一层次上的存储器的高速缓存。因此，

寄存器文件就是 L1 的高速缓存，而 L1 又是 L2 的高速缓存，L2 是主存的高速缓存，主存是磁盘的高速缓存。在某些带分布式文件系统的网络系统中，本地磁盘就是其他系统中磁盘上被存储数据的高速缓存。

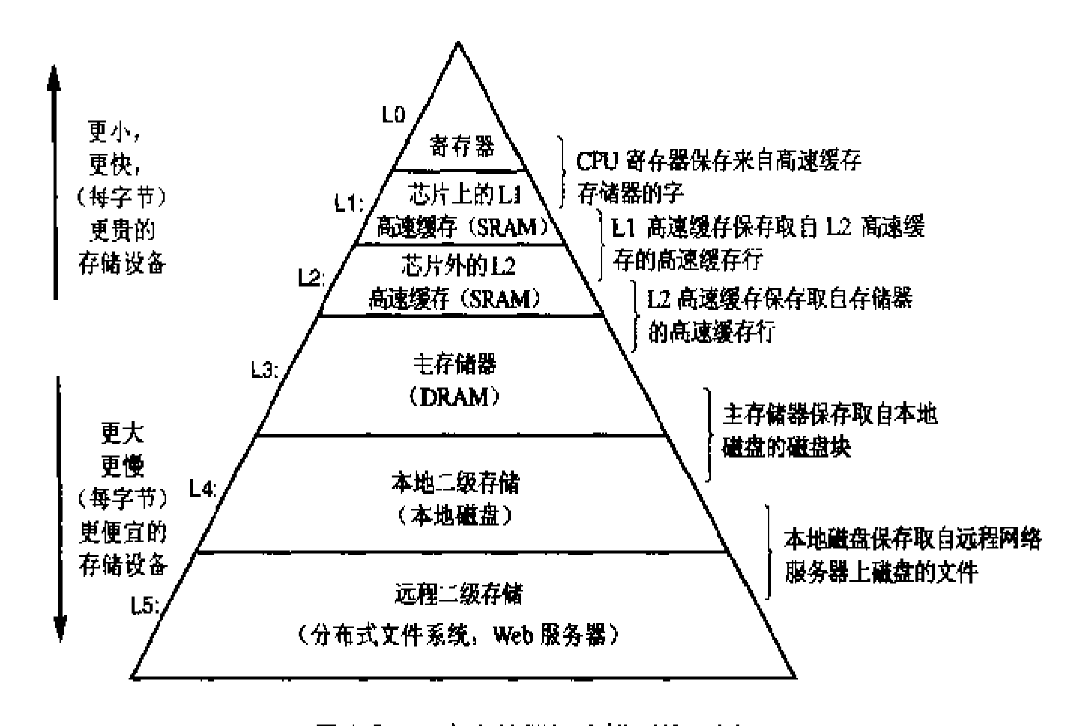


图 1.9 一个存储器层次模型的示例

就像程序员可以运用 L1 和 L2 的知识来提高程序性能一样，程序员同样可以利用对整个存储器层次模型的理解来提高程序性能。第 6 章将更详细地讨论这个问题。

1.7 操作系统管理硬件

让我们回到 hello 程序的例子。当 shell 加载和运行 hello 程序时，当 hello 程序输出自己的消息时，程序没有直接访问键盘、显示器、磁盘或者主存储器。取而代之的是，它们依靠操作系统提供的服务。我们可以把操作系统看成是应用程序和硬件之间插入的一层软件，如图 1.10 所示。所有应用程序对硬件的操作尝试都必须通过操作系统。

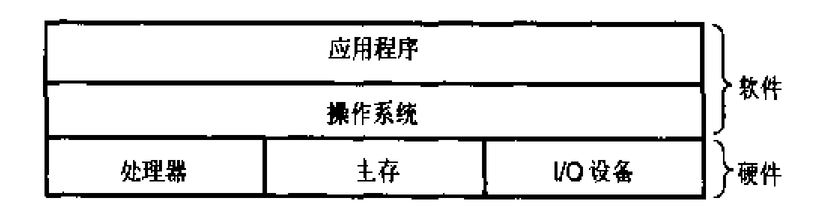


图 1.10 计算机系统的分层视图

操作系统有两个基本功能：防止硬件被失控的应用程序滥用；在控制复杂而又通常广泛不同的低级硬件设备方面，为应用程序提供简单一致的方法。操作系统通过图 1.11 中显示的几个基本的抽象概念（进程、虚拟存储器和文件）实现这两个功能。如图 1.11 所示，文件是对 I/O 设备的抽象表示，虚拟存储器是对主存和磁盘 I/O 设备的抽象表示，进程则是对处理器、主存和 I/O 设备的抽象表示。

我们将依次讨论每种抽象表示。

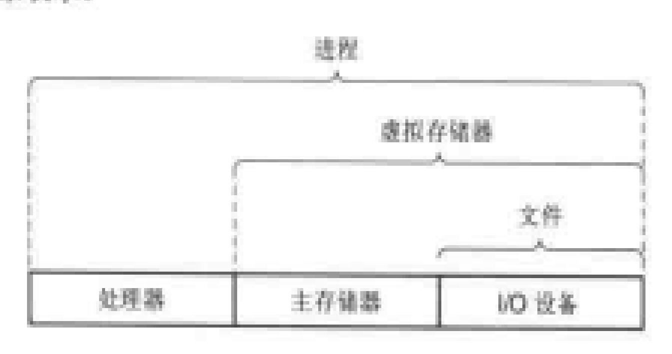


图 1.11 操作系统提供的抽象表示

旁注：Unix 和 Posix

20 世纪 60 年代是大型、复杂操作系统的年代，比如 IBM 的 OS/360 和 Honeywell 的 Multics 系统。OS/360 是历史上最成功的软件项目之一，而 Multics 持续了多年，却从来没有被广泛应用过。贝尔实验室曾经是 Multics 项目的最初参与者，但是因为考虑到该项目的复杂性和缺乏进展而于 1969 年退出。鉴于对 Mutics 项目不愉快的经验，一群贝尔实验室的研究人员——Ken Thompson、Dennis Ritchie、Doug McIlroy 和 Joe Ossanna 在 1969 年开始在 DEC PDP-7 计算机上完全用机器语言编写一个简单得多的操作系统。这个新系统中的很多思想，比如层次文件系统、作为用户级进程的 shell 概念，都是来自于 Multics，不过封装在更小、更简单的程序包里。1970 年，Brian Kernighan 给新系统命名为 "Unix"，也是双关语，暗指 "Multics" 的复杂性。1973 年其内核用 C 重新编写，1974 年，Unix 开始正式对外发布[65]。

因为贝尔实验室以宽松大方的条款向学校提供源代码，所以 Unix 在大专院校里获得了很多支持和发展。最有影响的工作发生在 20 世纪 70 年代晚期到 80 年代早期在美国加州大学伯克利分校，伯克利的研究人员在称为 Unix 4.xBSD（Berkeley Software Distribution）的一系列版本中增加了虚拟存储器和 Internet 协议。与此同时，贝尔实验室发布了他们自己的版本，也就是 System V Unix。其他厂商的版本，比如 Sun Microsystems 的 Solaris 系统，则是从这些原始的 BSD 和 System V Unix 版本中衍生而来的。

20 世纪 80 年代中期，Unix 厂商试图通过加入新的、一般不兼容的特性来使他们的程序与众不同，麻烦也就随之而来了。为了阻止这种趋势，IEEE（电气和电子工程师协会）发起努力来标准化 Unix，也就是后来 Richard Stallman 命名的 "Posix"。结果就得到了一系列的标准，称做 Posix 标准。这套标准覆盖了很多方面，比如 Unix 系统调用的 C 语言接口、shell 程序和工具、线程及网络编程。随着越来越多的系统越来越完全地遵从 Posix 标准，Unix 版本之间的差异正在逐渐消失。

1.7.1 进程

像 hello 这样的程序在现代系统上运行时，操作系统会提供一种假象，就好像系统上只有这个程序在运行，程序看上去独占地使用处理器、主存和 I/O 设备，而处理器看上去就像在不间断地一条接一条地执行程序中的指令。该程序的代码和数据就好像是系统存储器中惟一的对象。这些假象是通过进程的概念来实现的，进程是计算机科学中最重要和最成功的概念之一。

进程是操作系统对运行程序的一种抽象。在一个系统上可以同时运行多个进程，而每个进程都好

像在独占地使用硬件。我们称之为并发运行，实际上是说一个进程的指令和另一个进程的指令是交错执行的。操作系统实现这种交错执行的机制称为上下文切换（context switching）。

操作系统保存进程运行所需的所有状态信息。这种状态，也就是上下文（context），包括许多信息，比如PC和寄存器文件的当前值，以及主存的内容。在任何一个时刻，系统上都只有一个进程正在运行。当操作系统决定从当前进程转移控制权到某个新进程时，它就会进行上下文切换，即保存当前进程的上下文、恢复新进程的上下文，然后将控制权转移到新进程。新进程就会从它上次停止的地方开始。图1.12展示了我们的示例hello运行的基本场景。

在我们的示例场景中有两个同时运行的进程：shell进程和hello进程。最开始，只有shell进程在运行，等待命令行上的输入。当我们让它运行hello程序时，shell通过调用一个专门的函数，即系统调用，来执行我们的请求，系统调用会将控制权传递给操作系统。操作系统保存shell进程的上下文，创建一个新的hello进程及其上下文，然后将控制权传给新的hello进程。在hello进程终止后，操作系统恢复shell进程的上下文，并将控制权传回给它，它会继续等待下一命令行输入。

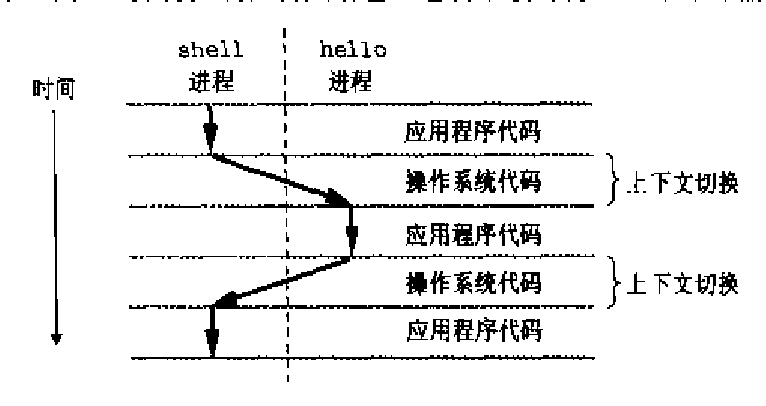


图1.12 进程的上下文切换

实现进程这个抽象概念需要低级硬件和操作系统软件的紧密合作。我们将在第8章中揭示这是如何工作的，以及应用程序是如何创建和控制它们的进程的。

进程这个抽象概念还暗示着由于不同的进程交错执行，打乱了时间的概念，使得程序员很难获得运行时间的准确和可重复测量。第9章讨论了现代系统中的各种时间概念，并描述了用来获得准确测量值的技术。

1.7.2 线程

尽管通常我们认为一个进程只有单一的控制流，但是在现代系统中，一个进程实际上可以由多个称为线程的执行单元组成，每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。由于网络服务器中对并行处理的要求，线程成为越来越重要的编程模型，因为多线程之间比多进程之间更容易共享数据，也因为线程一般都比进程更高效。在第13章中，你将学习到并行的基本概念，也包括线程化的概念。

1.7.3 虚拟存储器

虚拟存储器是一个抽象概念，它为每个进程提供了一个假象，好像每个进程都在独占地使用主存。每个进程看到的存储器都是一致的，称之为虚拟地址空间。图1.13所示的是Linux进程的虚拟地址空间（其他Unix系统的设计也与此类似）。在Linux中，最上面的四分之一的地址空间是预留给操作

系统中的代码和数据的，这对所有进程都一样。底部的四分之三的地址空间用来存放用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

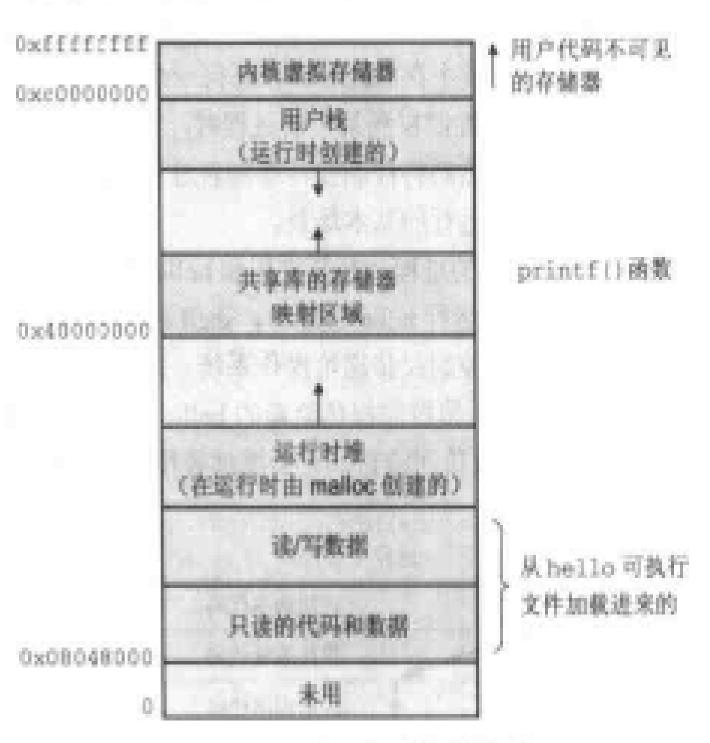


图 1.13 进程的虚拟地址空间

每个进程看到的虚拟地址空间由大量准确定义的区（area）构成，每个区都有专门的功能。在本书的后面你将学到更多的有关这些区的知识，但是先简单看看每一个区，从最低的地址开始，逐步向上研究将是非常有益的。

- **程序代码和数据**。代码是从同一固定地址开始，紧接着的是和 C 全局变量相对应的数据区。代码和数据区是由可执行目标文件直接初始化的，在我们的示例中就是可执行文件 hello。在第 7 章我们介绍链接和加载时，你会学习到更多有关地址空间中这部分的内容。
- **堆**。代码和数据区后紧随着的是运行时堆。代码和数据区是在进程一旦开始运行时就被指定了大小的，与此不同，作为调用像 malloc 和 free 这样的 C 标准库函数的结果，堆可以在运行时动态地扩展和收缩。在第 10 章学习管理虚拟存储器时，我们将更详细地研究堆。
- **共享库**。在地址空间的中间附近是一块用来存放像 C 标准库和数学库这样共享库的代码和数据的区域。共享库的概念非常强大，但是也是个相当难懂的概念。在第 7 章我们学习动态链接时，将学习共享库是如何工作的。
- **栈**。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。特别地，每次我们调用一个函数时，栈就会增长。每次我们从函数返回时，栈就会收缩。在第 3 章中你将学习编译器是如何使用栈的。
- **内核虚拟存储器**。内核是操作系统总是驻留在存储器中的部分。地址空间顶部的四分之一部分是为内核预留的。应用程序不允许读写这个区域的内容或者直接调用内核代码定义的函数。

虚拟存储器的运作需要硬件和操作系统软件间的精密复杂的互相合作，包括对处理器生成的每个地址的硬件翻译。基本思想是把一个进程虚拟存储器的内容存储在磁盘上，然后用主存作为磁盘的高

速缓存。第10章将解释它如何工作，以及它为什么对现代系统的运行如此重要。

1.7.4 文件

文件只不过就是字节序列。每个I/O设备，包括磁盘、键盘、显示器，甚至于网络，都可以被看成是文件。系统中的所有输入输出都是通过使用称为Unix I/O的一小组系统函数调用读写文件来实现的。

文件这个简单而精致的概念是非常强大的，因为它使得应用程序能够统一地看待系统中可能含有的所有各式各样的I/O设备。例如，处理磁盘文件内容的应用程序员可以非常幸福地无需了解具体的磁盘技术。进一步说，同一个程序可以在使用不同磁盘技术的不同系统上运行。你将在第11章中学习Unix I/O。

旁注：Linux项目

1991年8月，一个名为Linus Torvalds的芬兰研究生谨慎地发布了一个新的类Unix的操作系统内核。

来自：torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

新闻组：comp.os.minix

主题：在minix中你最想看到什么？

摘要：关于我的新操作系统的小调查

时间：1991年8月25日 20:57:08 GMT

每个使用minix的朋友，你们好－

我正在做一个（免费的）用在386（486）AT上的操作系统（只是业余爱好，它不会像GNU那样庞大和专业）。这个想法自从4月份就开始酝酿。我希望得到各位对minix喜欢和不满的反馈意见，因为我的操作系统在某些方面是模仿它的[其中包括相同的文件系统的物理设计（因为某些实际的原因）]。

我现在已经移植了bash（1.08）和gcc（1.40），并且看上去能运行。这意味着我需要几个月的时间来让它变得更实用一些，并且，我想要知道大多数人想要的特性。欢迎任何建议，但是我无法保证我能实现他们。:-)

Linus (torvalds@kruuna.helsinki.fi)

接下来的，如他们所说，就成为了历史。Linux逐渐发展成为一个技术和文化现象。通过和GNU项目的力量结合，Linux项目发展成为了一个完整的、符合Posix标准的Unix操作系统的版本，包括内核和所有支撑的基础设施。从手持设备到大型计算机，Linux在范围如此广泛的计算机上得到了应用。IBM的一个工作组甚至把Linux移植到了一块手表中！

1.8 利用网络系统和其他系统通信

系统漫游行之至此，我们一直是把系统视为一个孤立的硬件和软件的集合体。实际上，现代系统经常是通过网络和其他系统连接到一起的。从一个单独的系统来看，网络可被视为又一个I/O设备，如图1.14所示。当系统从主存拷贝一串字符到网络适配器时，数据流经过网络到达另一台机器，而不是到达本地磁盘驱动器。相似地，系统可以读取从其他机器发送来的数据，并把数据拷贝到自己的

主存。

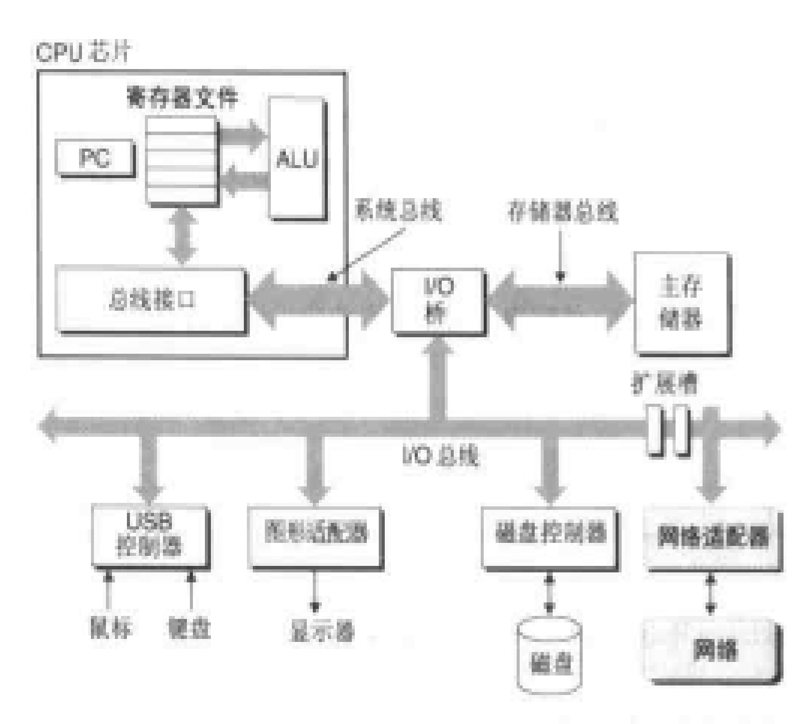


图1.14 网络也是一种I/O设备

随着像Internet这样的全球网络的出现，从一台主机拷贝信息到另外一台主机已经成为计算机系统最重要的用途之一。比如，像电子邮件、即时消息传送、万维网、FTP和telnet这样的应用都是基于通过网络拷贝信息的功能的。

回到我们的hello示例，我们可以使用熟悉的telnet应用在一个远程主机上运行hello程序。假设我们用本地主机上的telnet客户端连接远程主机上的telnet服务器。在我们登录到远程主机并运行shell后，远端的shell就在等待接收输入的命令。从这点上来看，在远端运行hello程序包括如图1.15所示的五个基本步骤。



图1.15 利用telnet跨越网络远程运行hello

当我们在telnet客户端键入"hello"串并敲下回车键后，客户端软件就会将这个字符串发送到telnet的服务器。在telnet服务器从网络上接收到这个串后，会把它传递给远端shell程序。接下来，远端shell运行hello程序，并将输出行返回给telnet服务器。最后，telnet服务器通过网络把输出串转发给telnet客户端，客户端就将输出串输出到我们的本地终端上。

这种在客户端和服务器之间交互的类型在所有的网络应用中是非常典型的。在第12章中，你将学会如何构造网络应用程序，并利用这些知识创建一个简单的Web服务器。

1.9 下一步

我们旋风式的系统漫游到此就结束了。从这次讨论中要得出一个很重要的观点，那就是系统不仅仅只是硬件。系统是互相交织的硬件和系统软件的集合体，它们必须共同协作以达到运行应用程序的最终目的。本书的余下部分将对这个论点进行展开。

1.10 小结

计算机系统是由硬件和系统软件组成的，它们共同协作以运行应用程序。计算机内部的信息被表示为一组组的位，它们依据不同的上下文又有不同的解释方式。程序被其他程序翻译成不同的形式，开始时是 ASCII 文本，然后被编译器和链接器翻译成二进制可执行文件。

处理器读取并解释存放在主存里的二进制指令。因为计算机花费了大量的时间在存储器、I/O 设备和 CPU 寄存器之间拷贝数据，所以系统中的存储设备就被按层次排列，CPU 寄存器在顶部，接着是多层的硬件高速缓存存储器、DRAM 主存储器和磁盘存储器。在层次模型中位于更高层的存储设备比低层的存储设备要快，单位比特造价也更高。程序员通过理解和运用这种存储层次结构的知识，可以优化他们 C 程序的性能。

操作系统内核是应用程序和硬件之间的媒介。它提供三个基本的抽象概念：文件是对 I/O 设备的抽象概念；虚拟存储器是对主存和磁盘的抽象概念；进程是处理器、主存和 I/O 设备的抽象概念。

最后，网络提供了计算机系统之间通信的手段。从某个系统的角度来看，网络就是一种 I/O 设备。

参考文献说明

Ritchie 写了关于早期 C 和 Unix 的有趣的第一手资料[63，64]。Ritchie 和 Thompson 提供了最早出版的 Unix 资料[65]。Silberschatz 和 Gavin[70]提供了关于 Unix 不同版本的详尽历史。GNU（www.gnu.org）和 Linux（www.linux.org）网页有大量的当前和历史信息。不幸的是，无法在线获得 Posix 标准，必须通过 IEEE（standards.ieee.org）定购。

第 1 部分

程序结构和执行

我们对计算机系统的探索是从学习计算机本身开始的，它由处理器和存储器子系统组成。在核心部分，我们需要方法来表示基本数据类型，比如整数和实数运算的近似值。然后，我们考虑机器级指令如何操作这样的数据，编译器如何将 C 程序翻译成这样的指令。接下来，我们研究几种实现处理器的方法，来更好地了解如何使用硬件资源来执行指令。一旦我们理解了编译器和机器级代码，我们就能通过编写可以更高效编译的源代码，来分析如何最大化程序的性能。我们以存储器子系统的设计来结束本部分，这是现代计算机系统最复杂的部分之一。

本书的这一部分将领着你深入了解应用程序是如何被表示和执行的。你将学会大量编程技巧，从而可以编写更可靠并充分利用计算资源的程序。

CHAPTER 2

信息的表示和处理

现代计算机存储和处理以二值信号表示的信息。这些普通的二进制数字，或者位（bit），形成了数字革命的基础。大家熟悉的使用了 1000 多年的十进制（以十为基数，base-10）起源于印度，在 12 世纪被阿拉伯数学家所改进，并在 13 世纪被意大利数学家 Leonardo Pisano（更有名的叫法是 Fibonacci）带到西方。使用十进制表示法对于有十个指头的人类来说是很自然的事情，但是当构造存储和处理信息的机器时，二进制值工作得更好。二值信号能够很容易地表示、存储和传输，例如，可以表示为穿孔卡片上有洞或无洞、导线上的高电压或低电压，或者磁场引起的顺时针或逆时针。基于二值信号的存储和执行计算的电子电路非常简单和可靠，使得制造商能够在一个单独的硅片上集成百万个这样的电路。

单独地来说，单个的位不是非常有用。然而，当我们把位组合在一起，再加上某种解释（interpretation），即给予不同的可能位模式以含意，我们就能够表示任何有限集合的元素。比如，使用一个二进制数字系统，我们能够用位组来编码非负数。通过使用标准的字符码，我们能够对一份文档中的字母和符号进行编码。在本章中，我们将讨论这两种编码，以及表示负数的编码和近似实数的编码。

我们考虑三种最重要的数字编码。*无符号*（unsigned）编码是基于传统的二进制表示法的，表示大于或者等于零的数字。*二进制补码*（two's-complement）编码是表示有符号整数的最常见的方式，有符号整数就是为正或者为负的数字。*浮点数*（floating-point）编码是表示实数的科学记数法的以二为基数的版本。计算机用这些不同的表示方法实现算术运算，例如加法和乘法，类似于相应的整数和实数运算。

计算机的表示法用有限的位数来对一个数字编码，因此，当结果太大以至不能表示时，某些运算就会溢出（overflow）。这会导致某些令人吃惊的后果。例如，在大多数今天的计算机上，计算表达式 200*300*400*500 会得出-884 901 888。这违背了整数运算的属性——计算一组正数的乘积产生了一个为负的结果。

另一方面，整数的计算机运算满足了真正整数运算的许多普通的属性。例如，乘法是可结合的和可交换的，这样一来计算下面任何一个 C 表达式，都会得出 884 901 888：

```
(500*400)*(300*200)
((500*400)*300)*200
((200*500)*300)*400
400*(200*(300*500))
```

计算机可能没有产生这个预期的结果，但是至少它是一致的！

浮点运算有完全不同的数学属性。虽然溢出会产生特殊的值+∞，但是一组正数的乘积总是正的。另一方面，由于表示的精度有限，浮点运算是不可结合的。例如，在大多数机器上，C 表达式（3.14+1e20）-1e20 求得的值会是 0.0，而 3.14+（1e20-1e20）求得的值会是 3.14。

通过研究实际数字的表示，我们能够了解可以表示的值的范围和不同算术运算的属性。对于编写在全部数值范围内都能正确工作，而且可以跨越不同机器、操作系统和编译器组合的可移植的程序来说，这种了解是非常重要的。

计算机用几种不同的二进制表示来编码数值。在第 3 章中随着你进入机器级编程，你将需要熟悉这些表示方式。在本章中，我们描述这些编码，并给你一些关于数字表示的推理练习。

通过直接操作位级的数字表示，我们得到了几种进行算术运算的方式。理解这些技术对于理解

编译算术表达式时产生的机器级代码是很重要的。

我们对这些内容的处理是非常精确的。我们从编码的基本定义开始，然后得出一些属性，例如可表示的数字的范围、它们的位级表示以及算术运算的属性。我们相信从这样一个抽象的观点来分析这些内容，对你来说是很重要的，因为程序员需要对计算机运算和更为人熟悉的整数和实数运算之间的关系有牢固的理解。尽管这看起来很吓人，但精确的处理只需要了解基本的代数知识。我们建议你将练习题作为巩固公式和一些实际生活例子之间联系的一种方法。

旁注：怎样阅读本章

如果你觉得等式和公式令人生畏，不要让它妨碍你学习本章的内容！为了完整性，我们提供全部的数学概念的推导，但是阅读这些内容的最好方法是在你首次阅读时跳过这些推导。相反，试着完成一些简单的示例（比如，练习题）来建立你的直觉，然后看看数学推导是如何巩固你的直觉的。

C++编程语言建立在C之上，使用完全相同的数字表示和运算。在本章中关于C的所有内容对C++都有效。另一方面，Java语言创造了一套新的数字表示和运算标准。C标准被设计为允许多种实现方式，而Java标准在数据的格式和编码上是详细而精确的。在本章中好几个地方我们都突出了Java支持的表示和运算。

2.1 信息存储

大多数计算机使用8位的块，或叫做字节（byte），来作为最小的可寻址的存储器单位，而不是访问存储器中单独的位。机器级程序将存储器视为一个非常大的字节数组，称为*虚拟存储器*（virtual memory）。存储器的每个字节都由一个惟一的数字来标识，称为它的*地址*（address），所有可能地址的集合就称为*虚拟地址空间*（virtual address space）。正如它的名字表明的，这个虚拟地址空间只是一个展现给机器级程序的概念性映像（image）。实际的实现（见第10章）使用的是随机访问存储器RAM、磁盘存储、特殊硬件和操作系统软件的结合，来为程序提供一个看上去统一的字节数组。

编译器和运行时系统的一个任务就是将这个存储器空间划分为更可管理的单元，来存放不同的*程序对象*（program object），也就是，程序数据、指令和控制信息。有各种机制可以用来分配和管理程序不同部分的存储。这种管理完全是在虚拟地址空间里完成的。例如，C中一个指针的值（无论它指向一个整数、一个结构或是某个其他程序单元）都是某个存储块的第一个字节的虚拟地址。C编译器还把每个指针和类型信息联系起来，这样它就可以根据指针值的类型，生成不同的机器级代码来访问存储在指针所指向位置处的值。尽管C编译器维护着这个类型信息，但是它生成的实际机器级程序并没有关于数据类型的信息。它简单地把每个程序对象视为一个字节块，而将程序本身看做一个字节序列。

给C语言初学者：C中指针的角色

指针是C的一个重要特性。它提供了引用数据结构的元素（包括数组）的机制。就像一个变量，指针也有两个方面：它的值和它的类型。它的值表示的是某个对象的位置，而它的类型表示那个位置上所存储对象的类型（比如，整数或者浮点数）。

2.1.1 十六进制表示法

一个字节包括 8 位。在二进制表示法中，它的值域是 00000000_2～11111111_2。如果看成十进制整数，它的值域就是 0_{10}～255_{10}。两种符号表示法对于描述位模式来说都不是非常方便。二进制表示法太冗长，而使用十进制表示法，与位模式的互相转化很麻烦。替代的方法是，我们以 16 为基数，或者叫做十六进制（hexadecimal）数，来书写位模式。十六进制（简写为“Hex”）使用数字“0”～“9”，以及字符“A”～“F”来表示 16 个可能的值。图 2.1 展示了 16 个十六进制数字对应的十进制值和二进制值。用十六进制书写，一个字节的取值范围为 00_{16}～FF_{16}。

在 C 中，以 0x 或 0X 开头的数字常量被认为是十六进制的值。字符“A”～“F”既可以是大写，也可以是小写。例如，我们可以将数字 $FA1D37B_{16}$ 写作 0xFA1D37B，或者 0xfa1d37b，甚至是大小写混合，比如，0xFa1D37b。在本书中，我们将使用 C 表示法来表示十六进制值。

十六进制数字	0	1	2	3	4	5	6	7
十进制值	0	1	2	3	4	5	6	7
二进制值	0000	0001	0010	0011	0100	0101	0110	0111

十六进制数字	8	9	A	B	C	D	E	F
十进制值	8	9	10	11	12	13	14	15
二进制值	1000	1001	1010	1011	1100	1101	1110	1111

图 2.1 十六进制表示法

每个十六进制数字都对 16 个值中的一个进行了编码。

编写机器级程序的一个常见任务就是手工地在位模式的十进制、二进制和十六进制表示之间转换。二进制和十六进制之间的转换是简单直接的，因为可以一次执行一个十六进制数字的转换。数字的转换可以参考图 2.1 所示的表。在你脑中做转换的一个简单的窍门是，记住十六进制数字 A、C 和 F 相应的十进制值。而对于把十六进制值 B、D 和 E 翻译成十进制值，则可以通过计算它们与前三个值的相对关系来完成。

比如，假设给你一个数字 0x173A4C。可以通过展开每个十六进制数字，将它转换为二进制格式，如下所示：

```
十六进制   1     7     3     A     4     C
二进制     0001  0111  0011  1010  0100  1100
```

这样就给出了二进制表示 000101110011101001001100。

反过来，如果给定一个二进制数字 1111001010110110110011，你可以通过首先把它分割为每四位一组，来把它转换为十六进制。不过要注意，如果位总数不是四的倍数，最左边的一组可以少于四位，前面用零补足。然后将每个四位组转换为相应的十六进制数字：

```
二进制     11    1100  1010  1101  1011  0011
十六进制   3     C     A     D     B     3
```

练习题 2.1

完成下面的数字转换:

A. 将 0x8F7A93 转换为二进制。

B. 将二进制 1011011110011100 转换为十六进制。

C. 将 0xC4E5D 转换为二进制。

D. 将二进制 1101011011011111100110 转换为十六进制。

当值 x 是 2 的幂时，也就是，对于某个 n，$x=2^n$，我们可以很容易地将 x 写成十六进制形式，只要记住 x 的二进制表示就是 1 后面跟 n 个零。十六进制数字 0 代表四个二进制 0。所以，对于被写成 $i+4j$ 形式的 n 来说，其中 $0 \le i \le 3$，我们可以把 x 写成开头的十六进制数字为 1（i=0）、2（i=1）、4（i=2）或者 8（i=3），后面跟随着 j 个十六进制的 0。比如，$x=2048=2^{11}$，我们有 $n = 11 = 3 + 4 \cdot 2$，从而得到十六进制表示 0x800。

练习题 2.2

填写下表中的空白项，给出 2 的不同次幂的二进制和十六进制表示:

n	2^n（十进制）	2^n（十六进制）
11	2048	0x800
7		
	8192	
		0x2000
16		
	256	
		0x20

十进制和十六进制表示之间的转换需要使用乘法或者除法来处理一般情况。将一个十进制数字 x 转换为十六进制，我们可以反复地用 16 除 x，得到一个商 q 和一个余数 r，也就是 $x = q \cdot 16 + r$。然后，我们用十六进制数字表示的 r 作为最低位数字，并且通过对 q 反复进行这个过程得到剩下的数字。例如，考虑十进制 314156 的转换:

$$
\begin{aligned}
314156 &= 19634 \cdot 16 + 12 \quad &(C)\\
19634 &= 1227 \cdot 16 + 2 \quad &(2)\\
1227 &= 76 \cdot 16 + 11 \quad &(B)\\
76 &= 4 \cdot 16 + 12 \quad &(C)\\
4 &= 0 \cdot 16 + 4 \quad &(4)
\end{aligned}
$$

从这里，我们能读出十六进制表示为 0x4CB2C。

反过来，将一个十六进制数字转换为十进制数字，我们可以用相应的 16 的幂乘以每个十六进制数字。比如，给定数字 0x7AF，我们计算它对应的十进制值为 $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967$。

练习题 2.3

一个字节可以被表示为两个十六进制数字。填写下表中缺失的项，给出不同字节模式的十进制、二进制和十六进制值：

十进制	二进制	十六进制
0	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

旁注：十进制和十六进制间的转换

为了在十进制和十六进制之间转换较大的数值，最好是让计算机或者计算器来完成这项工作。例如，下面的 Perl 语言脚本将一列数字从十进制转换为十六进制：

code/data/d2h

```
1  #!/usr/local/bin/perl
2  # Convert list of decimal numbers into hex
3
4  for ($i = 0; $i < @ARGV; $i++) {
5  printf("%d\t= 0x%x\n", $ARGV[$i], $ARGV[$i]);
6  }
```

code/data/d2h

一旦这个文件被设置为可执行的，命令：

```
unix> ./d2h 100 500 751
```

会产生输出：

```
100=0x64
500=0x1f4
751=0x2ef
```

相似地，下面的脚本将十六进制转换为十进制：

code/data/h2d

```
1  #!/usr/local/bin/perl
2  # Convert list of hex numbers into decimal
3
```

```
4 for ($i = 0; $i < @ARGV; $i++) {
5   $val = hex($ARGV[$i]);
6   printf("0x%x = %d\n", $val, $val);
7 }
```

code/data/h2d

练习题 2.4

不将数字转换为十进制或者二进制，试着解答下面的算术题，答案要用十六进制表示。提示：只要修改你执行十进制加法和减法所使用的方法，以 16 为基数。

A. 0x502c+0x8=

B. 0x502c−0x30=

C. 0x502c+64=

D. 0x50da−0x502c=

2.1.2 字

每台计算机都有一个字长（word size），指明整数和指针数据的标称大小（nominal size）。因为虚拟地址是以这样的字来编码的，所以字长决定的最重要的系统参数就是虚拟地址空间的最大大小。也就是说，对于一个字长为 n 位的机器而言，虚拟地址的范围为 $0\sim2^n-1$，程序最多访问 2^n 字节。

今天大多数计算机的字长都是 32 位。这就限制了虚拟地址空间为 4 千兆字节（写作 4GB），也就是说，刚刚超过 4×10^9 字节。虽然对大多数应用而言，这个空间足够大了，但是现在已经有许多大型的科学和数据库应用需要更大的存储了。因此，随着存储器价格的降低，字长为 64 位的高端机器正逐渐变得普遍起来。

2.1.3 数据大小

计算机和编译器使用不同的方式来编码数字，比如不同长度的整数和浮点数，从而支持多种数字格式。比如，许多机器都有处理单个字节的指令，也有处理表示为两字节、四字节或者八字节整数的指令，还有些指令支持表示为四字节和八字节的浮点数。

C 语言支持整数和浮点数的多种数据格式。C 的数据类型 char 表示一个单独的字节。尽管"char"这个名字是由于它被用来存储文本串中的单个字符这一事实而来的，但它也能被用来存储整数值。C 的数据类型 int 之前还能加上限定词 long 和 short，提供各种大小的整数表示。图 2.2 展示了为各种 C 数据类型分配的字节数。准确的字节数依赖于机器和编译器。我们展示了两个有代表性的例子：典型的 32 位机器和 Compaq Alpha 体系结构，其中 Compaq Alpha 是针对高端应用的 64 位机器。大多数 32 位机器使用"典型"的分配方式。可以观察到，"短"整数分配有两字节，而不加限制的 int 为四字节，"长"整数使用机器的全字长。

图 2.2 也说明了指针（例如，一个被声明为类型为"char *"的变量）使用机器的全字长。大多数机器还支持两种不同的浮点格式：单精度（在 C 中声明为 float）和双精度（在 C 中声明为 double）。这些格式分别使用四字节和八字节。

C 声明	典型的32位机器	Compaq Alpha 机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

图 2.2 C 语言中数字数据类型的大小（以字节为单位）

分配的字节数随着机器和编译器的不同而不同。

给 C 语言初学者：声明指针

对于任何数据类型 T，声明

```
T *p;
```

表明 p 是一个指针变量，指向类型 T 的一个对象。例如

```
char *p;
```

就将一个指针声明为指向 char 类型的一个对象。

程序员应该力图使他们的程序在不同的机器和编译器上可移植。可移植性的一个方面就是使程序对不同数据类型的确切大小不敏感。C 标准对不同数据类型的数字范围设置了下界，这点在后面还将讲到，但是却没有上界。因为 32 位机器在过去 20 年里一直是标准，许多程序的编写都是以图 2.2 中"典型的 32 位机器"列出的分配原则为假设的。在不久的将来，随着 64 位机器越来越重要，在将这些程序移植到新机器上时，许多隐藏的对字长的依赖就会显现出来，成为错误。比如，许多程序员假设一个声明为 int 类型的程序对象能被用来存储一个指针。这在大多数 32 位的机器上工作正常，但是在一台 Alpha 机器上却会导致问题。

2.1.4 寻址和字节顺序

对于跨越多字节的程序对象，我们必须建立两个规则：这个对象的地址是什么和我们在存储器中如何对这些字节排序。在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节序列中最小的地址。例如，假设一个类型为 int 的变量 x 的地址为 0x100，也就是说，地址表达式&x 的值为 0x100。那么，x 的四字节将被存储在存储器的 0x100、0x101、0x102 和 0x103 位置。

对表示一个对象的字节序列排序，有两个通用的规则。考虑一个 w 位的整数，有位表示$[x_{w-1}, x_{w-2}, \cdots, x_1, x_0]$，其中 x_{w-1} 是最高有效位，而 x_0 是最低有效位。假设 w 是 8 的倍数，这些位就能被分组成为字节，其中最高有效字节包含位$[x_{w-1}, x_{w-2}, \cdots, x_{w-8}]$，而最低有效字节包含位$[x_7, x_6, \cdots, x_0]$，其他字节包含中间的位。某些机器选择在存储器中按照从最低有效字节到最高有效字节的顺序存储对象，而另一些机器则按照从最高有效字节到最低有效字节的顺序存储。前一种规则——最低有效字节在最前面的方式被称为小端法（little endian）。大多数源自以前的 Digital Equipment 公司（现在是 Compaq 公司的一部分）的机器，以及 Intel 的机器都采用这种规则。后一种规则（最高有效字

节在最前面的方式）被称为大端法（big endian）。IBM、Motorola 和 Sun Microsystems 的大多数机器都采用这种规则。注意我们说的是“大多数”。这些规则并没有严格按照企业界限来划分。比如，IBM 制造的个人计算机使用的是 Intel 兼容的处理器，因此就是小端法。许多微处理器芯片，包括 Alpha 和 Motorola 的 PowerPC，能够运行在任一种模式中，其取决于芯片加电启动时确定的字节顺序规则。

继续我们前面的示例，假设变量 x 类型为 int，位于地址 0x100 处，有一个十六进制值为 0x01234567。地址范围 0x100～0x103 的字节顺序依赖于机器的类型：

大端法

	0x100	0x101	0x102	0x103	
…	01	23	45	67	…

小端法

	0x100	0x101	0x102	0x103	
…	67	45	23	01	…

注意，在字 0x01234567 中，高位字节的十六进制值为 0x01，而低位字节值为 0x67。

令人吃惊的是，在哪种字节顺序是合适的这个问题上，人们表现得非常情绪化。实际上，术语“little endian（小端）”和“big endian（大端）”来自于 Jonathan Swift 的《格利佛游记（Gulliver's Travels）》，其中交战的两个派别无法就应该从哪一端——小端还是大端——打开一个半熟的鸡蛋达成一致。就像鸡蛋的问题一样，没有技术原因来选择字节顺序规则，因此争论退化成为关于社会政治论题的口角。对于哪种字节排序的选择是任意的。

旁注：“endian”的起源

下面就是 Jonathan Swift 在 1726 年如何描述大、小端之争的历史的：

“……我下面要告诉你的是，Lilliput 和 Blefuscu 这两大强国在过去三十六个月里一直在苦战。战争开始是由于以下的原因：我们大家都认为，吃鸡蛋前，原始的方法是打破鸡蛋较大的一端，可是当今皇帝的祖父小时候吃鸡蛋，一次按古法打鸡蛋时碰巧将一个手指弄破了，因此他的父亲，当时的皇帝，就下了一道敕令，命令全体臣民吃鸡蛋时打破鸡蛋较小的一端，违令者重罚。老百姓们对这项命令极为反感。历史告诉我们，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。这些叛乱大多都是由 Blefuscu 的国王大臣们煽动起来的。叛乱平息后，流亡的人总是逃到那个帝国去寻救避难。据估计，先后几次有一万一千人情愿受死也不肯去打破鸡蛋较小的一端。关于这一争端，曾出版过几百本大部著作，不过大端派的书一直是受禁的，法律也规定该派的任何人不得做官。”（此段译文摘自网上蒋剑锋译的《格利佛游记》第一卷第 4 章。）

在他那个时代，Swift 是在讽刺英国（Lilliput）和法国（Blefuscu）之间持续的冲突。Danny Cohen，一位网络协议的早期开创者，第一次使用这两个术语来指代字节顺序[17]，后来这个术语被广泛接纳了。

对于大多数应用程序员来说，他们机器的字节顺序是完全不可见的。无论为哪种类型的机器所编译的程序都会得到同样的结果。不过有时候，字节顺序会成为问题。首先是在不同类型的机器之间通过网络传送二进制数据时。一个常见的问题是当小端法机器产生的数据被发送到大端法机器或

者反之时，接收程序会发现，字里的字节成了反序的。为了避免这类问题，网络应用程序的代码编写必须遵守已建立的关于字节顺序的规则，以确保发送方机器将它的内部表示转换成网络标准，而接收方机器则将网络标准转换为它的内部表示。我们将在第12章中看到这种转换的例子。

字节顺序变得重要的第二种情况是当阅读表示整数数据的字节序列时。这通常发生在检查机器级程序时。作为一个示例，从某个文件中摘出了下面这行代码，该文件给出了一个针对Intel处理器的机器级代码的文本表示：

```
80483bd: 01 05 64 94 04 08          add   %eax, 0x8049464
```

这一行是由反汇编器（disassembler）生成的，反汇编器是一种确定可执行程序文件所表示的指令序列的工具。我们将在下一章中学习有关这些工具的更多知识，以及怎样解释像这样的行。而现在，我们只是注意这行表述了十六进制字节串01 05 64 94 04 08是一条指令的字节级表示，这条指令是增加一个字宽的数据到存储在主存地址0x8049464的值上。如果我们取出这个序列的最后四字节：64 94 04 08，并且按照相反的顺序写出，我们得到08 04 94 64。去掉开头的零，我们就得到值0x8049464，就是右边写着的数值。当阅读像此例中一样的小端法机器生成的机器级程序表示时，经常会将字节按照相反的顺序显示。书写字节序列的自然方式是最低位字节在左边，而最高位字节在右边，但是这和书写数字时最高有效位在左边，最低有效位在右边的通常方式是相反的。

字节顺序变得可见的第三种情况是当编写规避正常的类型系统的程序时。在C语言中，可以通过使用强制类型转换（cast）来允许以一种不同于它被创造时的数据类型来引用一个对象。大多数应用编程都强烈不推荐这种编码技巧，但是它们对系统级编程来说是非常有用，甚至是必须的。

图2.3展示了一段C代码，它使用强制类型转换来访问和打印不同程序对象的字节表示。我们用typedef将数据类型byte_pointer定义为一个指向类型为“unsigned char”的对象的指针。这样一个字节指针引用一个字节序列，其中每个字节都被认为是一个非负整数。第一个例程show_bytes的输入是一个字节序列的地址（它用一个字节指针来指示）和一个字节数。show_bytes打印出以十六进制表示的字节。C格式化指令“%.2x”表示整数必须用至少两个数字的十六进制格式输出。

给C语言初学者：使用typedef来命名数据类型

C中的typedef声明提供了一种给数据类型命名的方式。这能够极大地改善代码的可读性，因为深度嵌套的类型声明很难读懂。

typedef的语法与声明变量十分相像，除了它使用的是类型名，而不是变量名。因此，图2.3中byte_pointer的声明和将一个变量声明为类型“unsigned char”有相同的形式。

例如，声明：

```
typedef int *int_pointer;
int_pointer ip;
```

将类型“int_pointer”定义为一个指向int的指针，并且声明了这种类型的变量ip。我们还可以直接声明这个变量为：

```
int *ip;
```

code/data/show-bytes.c

```
1 #include <stdio.h>
2
```

```
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes(byte_pointer start, int len)
6 {
7 int i;
8 for (i = 0; i < len; i++)
9 printf(" %.2x", start[i]);
10 printf("\n");
11 }
12
13 void show_int(int x)
14 {
15 show_bytes((byte_pointer) &x, sizeof(int));
16 }
17
18 void show_float(float x)
19 {
20 show_bytes((byte_pointer) &x, sizeof(float));
21 }
22
23 void show_pointer(void *x)
24 {
25 show_bytes((byte_pointer) &x, sizeof(void *));
26 }
```

code/data/show-bytes.c

图 2.3 打印程序对象的字节表示

这段代码使用强制类型转换来规避类型系统。

给 C 语言初学者：使用 printf 格式化输出

printf 函数（还有它的同类 fprintf 和 sprintf）提供了对格式化细节有相当大控制的输出信息的方式。第一个参数是格式串，而其余的参数都是要打印的值。在格式串里，每个以"%"开始的字符序列都表示如何格式化下一个参数。典型的示例包括"%d"是输出一个十进制整数，"%f"是输出一个浮点数，而"%c"是输出一个字符，这个字符的编码是由参数给出的。

给 C 语言初学者：指针和数组

在函数 show_bytes（图 2.3）中，我们看到指针和数组之间紧密的联系，这将在 3.8 节中详细描述。我们看到这个函数有一个类型为 byte_pointer（被定义为一个指向 unsigned char 的指针）的参数 start，但是我们在第 9 行看到数组引用 start[i]。在 C 中，我们能够用数组表示法来引用指针，同时我们也能用指针表示法来引用数组元素。在这个例子中，引用 start[i]表示我们想要读取以 start 指向的位置为起始的第 i 个位置处的字节。

过程 show_int、show_float 和 show_pointer 展示了如何使用程序 show_bytes 来分别输出类型为 int、float 和 void *的 C 程序对象的字节表示。可以观察到它们仅仅传递给 show_bytes 一个指向它们

参数 *x* 的指针&*x*，且这个指针被强制类型转换为“unsigned char *”。这种强制类型转换告诉编译器，程序应该把这个指针看成指向一个字节序列，而不是指向一个原始数据类型的对象。然后，这个指针将指向对象使用的最低字节地址。

给 C 语言初学者：指针的创建和间接引用

在图 2.3 的第 15、20 和 25 行，我们看到对 C 和 C++中两种独有操作的使用。C 的“取地址”运算符&创建一个指针。在这三行中，表达式&x 创建了一个指向保存变量 x 的位置的指针。这个指针的类型取决于 x 的类型，因此这三个指针的类型分别为 int*、float*和 void。（数据类型 void*是一种特殊类型的指针，没有相关的类型信息。）**

强制类型转换运算符是将一种数据类型转换为另一种。因此，强制类型转换 (byte_pointer) &x 表明无论指针&x 以前是什么类型，它现在就是一个指向数据类型为 unsigned char 的指针了。

这些过程使用 C 的运算符 sizeof 来确定对象使用的字节数。一般来说，表达式 sizeof(*T*)返回存储一个类型为 *T* 的对象所需要的字节数。使用 sizeof，而不是一个固定的值，是向编写在不同机器类型上可移植的代码迈进了一步。

在几种不同的机器上运行如图 2.4 所示的代码，得到如图 2.5 所示的结果。使用了以下机器：

Linux： Intel Pentium II 运行 Linux。

NT： Intel Pentium II 运行 Windows-NT。

Sun： Sun Microsystems UltraSPARC 运行 Solaris。

Alpha： Compaq Alpha 21164 运行 Tru64 Unix。

code/data/show-bytes.c

```
1 void test_show_bytes(int val)
2 {
3     int ival = val;
4     float fval = (float) ival;
5     int *pval = &ival;
6     show_int(ival);
7     show_float(fval);
8     show_pointer(pval);
9 }
```

code/data/show-bytes.c

图 2.4　字节表示的示例

这段代码输出了样本数据对象的字节表示。

我们的参数 12 345 的十六进制表示为 0x00003039。对于 int 类型的数据，除了字节顺序以外，我们在所有机器上都得到相同的结果。特别地，我们可以看到在 Linux、NT 和 Alpha 上，最低有效字节值 0x39 最先输出，这说明它们是小端法机器，而在 Sun 上最后输出，这说明 Sun 是大端法机器。同样地，float 类型的数据，除了字节顺序以外，也都是相同的。另一方面，指针值却是完全不同的。不同的机器/操作系统配置使用不同的存储分配规则。一个值得注意的特性是 Linux 和 Sun 的机器使用四字节地址，而 Alpha 使用八字节地址。

机器	值	类型	字节（十六进制）
Linux	12 345	int	39 30 00 00
NT	12 345	int	39 30 00 00
Sun	12 345	int	00 00 30 39
Alpha	12 345	int	39 30 00 00
Linux	12 345.0	float	00 e4 40 46
NT	12 345.0	float	00 e4 40 46
Sun	12 345.0	float	46 40 e4 00
Alpha	12 345.0	float	00 e4 40 46
Linux	&ival	int *	3c fa ff bf
NT	&ival	int *	1c ff 44 02
Sun	&ival	int *	ef ff fc e4
Alpha	&ival	int *	80 fc ff 1f 01 00 00 00

图 2.5 不同数据值的字节表示

除了字节顺序以外，int 和 float 的结果是一样的。指针值是与机器相关的。

可以观察到，尽管浮点和整型数据都是对数值 12 345 编码，但是它们有非常不同的字节模式：整型为 0x00003039，而浮点数为 0x4640E400。一般而言，这两种格式使用不同的编码方法。如果我们将这些十六进制模式扩展为二进制形式，并且适当地将它们移位，我们就会发现一个有 13 个相匹配的位的序列，如下面的一串星号标识出来的那样：

```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
                     *************
             4   6   4   0   E   4   0   0
            01000110010000001110010000000000
```

这并不是巧合。当我们研究浮点格式时，还将再回到这个例子。

练习题 2.5

思考下面对 show_bytes 的三个调用：

```
int val = 0x12345678;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

指出在小端法机器和大端法机器上，每个调用的输出值。

A. 小端法：　　　　　　大端法：

B. 小端法：　　　　　　大端法：

C. 小端法：　　　　　　大端法：

练习题 2.6

使用 show_int 和 show_float，我们确定整数 3490593 的十六进制表示为 0x00354321，而浮点数

3490593.0 的十六进制表示为 0x4A550C84。

A. 写出这两个十六进制值的二进制表示。

B. 移动这两个二进制串的相对位置，使得它们相匹配的位数最大。

C. 有多少位相匹配呢？串中的什么部分不相匹配？

2.1.5 表示字符串

C 中的字符串被编码为一个以 null（其值为零）字符结尾的字符数组。每个字符都由某个标准编码来表示，最常见的就是 ASCII 字符码。因此，如果我们以参数“12345”和 6（包括终止符）来运行例程 show_bytes，我们得到结果 31 32 33 34 35 00。请注意，十进制数字 *x* 的 ASCII 码正好是 0x3*x*，而终止字节的十六进制表示为 0x00。在使用 ASCII 码作为字符码的任何系统上都将得到相同的结果，与字节顺序和字大小规则无关。因而，文本数据比二进制数据具有更强的平台独立性。

旁注：生成一张 ASCII 表

可以通过执行命令 man ascii 来得到一张 ASCII 字符码的表。

练习 2.7

下面对 show_bytes 的调用将输出什么结果？

```
char *s = "ABCDEF";
show_bytes(s, strlen(s));
```

注意字母“A”~“Z”的 ASCII 码为 0x41～0x5A。

旁注：Unicode（统一字符编码标准）字符集

ASCII 字符集适合于编码英语文档，但是它在表达一些特殊字符方面并没有太多办法，例如法语的“Ç”。它完全不适合编码希腊语、俄语和中文这样语言的文档。最近，16 位的 Unicode 字符集被采纳用来支持所有语言的文档。这种双字节字符集表示使得大量不同字符的表示变为可能。Java 编程语言使用 Unicode 来表示字符串。对于 C 也有可用的程序库来提供 Unicode 版本的标准字符串函数，例如 strlen 和 strcpy。

2.1.6 表示代码

考虑下面的 C 函数：

```
1 int sum(int x, int y)
2 {
3     return x + y;
4 }
```

当在我们的示例机器上编译时，我们生成有如下字节表示的机器代码：

```
Linux:   55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
NT:      55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
Sun:     81 C3 E0 08 90 02 00 09
Alpha:   00 00 30 42 01 80 FA 6B
```

这里我们发现除了 NT 和 Linux 机器以外，指令编码是不同的。不同的机器类型使用不同的且不兼容的指令和编码方式。NT 和 Linux 机器使用的都是 Intel 处理器，因此支持相同的机器级指令。然而，一般而言，一个可执行的 NT 程序和一个 Linux 程序的结构是不同的，因此这些机器并不完全是二进制兼容的。二进制代码很少能在不同机器和操作系统组合之间移植。

计算机系统的一个基本概念就是从机器的角度来看，程序仅仅只是字节序列。机器没有关于原始源程序的任何信息，除了可能有些用来帮助调试的辅助表以外。当我们在第 3 章中学习机器级编程时，将更清楚地了解这一点。

2.1.7 布尔代数和环

因为二进制值是计算机编码、存储和操作信息的核心，所以围绕数值 0 和 1 已经演化出了丰富的数学知识体系。这起源于 1850 年左右乔治・布尔（George Boole）的工作，因此也被称为布尔代数（Bool algebra）。布尔观察到通过将二进制值 1 和 0 编码为逻辑值 TRUE（真）和 FALSE（假），能够设计出一种代数，研究命题逻辑的属性。

存在大量不同的布尔代数，其中最简单的是定义在二元素集合{0，1}基础上的运算。图 2.6 定义了这种布尔代数中的几种运算。我们用来表示这些运算的符号是和 C 的位级运算使用的符号相匹配的，这些将在后面讨论到。布尔运算~对应于逻辑运算 NOT，在命题逻辑中表示为¬。也就是说，当 P 不是真的时候，我们就说 $\neg P$ 是真的，反之亦然。相应地，当 P 等于 0 时，~P 等于 1，反之亦然。布尔运算&对应于逻辑运算 AND，在命题逻辑中表示为∧。当 P 和 Q 都为真时，我们说 $P \wedge Q$ 成立，反之亦然。相应地，只有当 p=1 且 q=1 时，p&q 才等于 1。布尔运算 | 对应于逻辑运算 OR，在命题逻辑中表示为∨。当 P 或者 Q 为真时，我们说 $P \vee Q$ 成立。相应地，当 p=1 或者 q=1 时，p | q 等于 1。布尔运算^对应于逻辑运算 EXCLUSIVE-OR（异或），在命题逻辑中表示为 ⊕。当 P 或 Q 为真但不同为真时，我们说 $P \oplus Q$ 成立。相应地，当 p=1 且 q=0，或者 p=0 且 q=1 时，p^q 等于 1。

<table>
<tr><th>~</th><th></th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>

<table>
<tr><th>&</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
</table>

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

<table>
<tr><th>^</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>

图 2.6 布尔代数的运算

二进制值 1 和 0 表示逻辑值 TRUE 或者 FALSE，而运算符~、&、| 和∧依次表示逻辑运算 NOT、AND、OR 和 EXCLUSIVE-OR。

后来创立信息理论领域的 Claude Shannon 首先建立了布尔代数和数字逻辑之间的联系。在他 1937 年的硕士论文中，他表明了布尔代数可以用来设计和分析机电继电器网络。尽管从那时起计算机技术已经取得了相当的发展，但是布尔代数仍然在数字系统的设计和分析中扮演着重要的角色。

在整数运算和布尔代数之间有许多相似点，同时也有一些重要的不同之处。特别地，整数集合，用 Z 来表示，形成了一个称为环的数据结构，表示为<Z，+，×，−，0，1>，其中加法为求和运算，乘法为求积运算，负号作为加法的逆运算，而元素 0 和 1 作为加法和乘法的单位元。布尔代数<{0,1}, |, &, ~, 0, 1>有相似的属性。图 2.7 突出了两种结构的属性，展示了两者的共同点和各自独特的属性。一个重要的不同之处就在于~a 不是 a 在|运算下的逆元。

共有属性

属　　性	整数环	布尔代数
交换性	$a+b=b+a$ $a\times b=b\times a$	$a\|b=b\|a$ $a\&b=b\&a$
结合性	$(a+b)+c=a+(b+c)$ $(a\times b)\times c=a\times(b\times c)$	$(a\|b)\|c=a\|(b\|c)$ $(a\&b)\&c=a\&(b\&c)$
分配性	$a\times(b+c)=(a\times b)+(a\times c)$	$a\&(b\|c)=(a\&b)\|(a\&c)$
同一性	$a+0=a$ $a\times 1=a$	$a\|0=a$ $a\&1=a$
消除性	$a\times 0=0$	$a\&0=0$
相消性	$-(-a)=a$	$\tilde{}(\tilde{}a)=a$

环的特性

相逆性	$a+-a=0$	—

布尔代数的特性

分配性	—	$a\|(b\&c)=(a\|b)\&(a\|c)$
相补性	—	$a\|\tilde{}a=1$ $a\&\tilde{}a=0$
幂等性	—	$a\&a=a$ $a\|a=a$
吸收性	—	$a\|(a\&b)=a$ $a\&(a\|b)=a$
DeMorgan 定律	—	$\tilde{}(a\&b)=\tilde{}a\|\tilde{}b$ $\tilde{}(a\|b)=\tilde{}a\&\tilde{}b$

图 2.7　整数环和布尔代数的比较

两种数学结构有很多共同的属性，但也有一些关键的不同点，尤其是在-和~之间。

旁注：抽象代数有什么好处？

抽象代数包括识别和分析不同领域内数学运算的共同属性。典型地，一个代数就是被定义为一组元素、一些关键运算和一些重要的元素。比如，模数运算也构成一个环。对于模数 n，代数被表示为 $\langle Z_n, +_n, \times_n, -_n, 0, 1\rangle$，其中各个部分定义如下：

$$
\begin{aligned}
Z_n &= \{0,1,\cdots,n-1\} \\
a +_n b &= a+b \bmod n \\
a \times_n b &= a\times b \bmod n \\
-_n a &= \begin{cases} 0, & a=0 \\ n-a, & a>0 \end{cases}
\end{aligned}
$$

虽然模数运算和整数运算得到的结果不同，但是它们还是有很多相同的数学属性的。其他知名的环还包括有理数和实数。

如果我们用 EXCLUSIVE-OR 运算来取代布尔代数中的 OR 运算，并且用同一运算（identity operation）I 来取代补运算~，这里对于所有的 a，$I(a)=a$，那么我们就得到一个结构<{0, 1}, ^, &, I, 0, 1>。这个结构不再是一个布尔代数，实际上它是一个环。它能被视为一种特别简单的环的形式，是由所有整数{0，1，…，n-1}组成的，并且加法和乘法都是模 n 的。在这个例子中，我们设 n=2。也就是说，布尔 AND 和 EXCLUSIVE-OR 分别相当于模 2 的乘法和加法。这个代数的一个奇怪的属性就是，每个元素都是它自己的加法逆元：a^$I(a)$=a^a=0。

旁注：除了数学家，还有谁关心布尔环？

每当你享受一张 CD 记录的清晰的音乐或者一张 DVD 记录的高质量的视频画面时，你就在利用布尔环。这些技术都依赖于纠错码从碟片上可靠地获取位，即使碟片很脏，甚至有划痕。这些纠错码的数学基础就是基于布尔环的线性代数。

我们能够将这四种布尔运算扩展到位向量上，位向量就是某个固定长度为 w 的 0 或 1 的串。我们通过将运算应用到参数相匹配的元素上，来定义对位向量的运算。例如，我们定义[a_{w-1}，a_{w-2}，…，a_0]&[b_{w-1}，b_{w-2}，…，b_0]为[a_{w-1}&b_{w-1}，a_{w-2}&b_{w-2}，…，a_0&b_0]，对于运算~、| 和^有类似的定义。设$\{0, 1\}^w$表示所有长度为 w 的 0、1 串的集合，而 a^w 表示由 w 个符号 a 组成的串，那么你能看到得到这样的代数：<$\{0, 1\}^w$，|，&，~，0^w，1^w>和<$\{0, 1\}^w$，^，&，I，0^w，1^w>，分别构成了布尔代数和环。每一个不同的 w 值就定义了一个不同的布尔代数和一个不同的布尔环。

旁注：布尔环和模运算一样吗？

二元的布尔环<{0，1}，^，&，I，0，1>与整数模 2 环<Z_2，$+_2$，$\times_2$，$-_2$，0，1>是相同的。然而，推广到长度为 w 的位向量，会得到与模数运算非常不同的环。

练习题 2.8

填写下表，给出对位向量的布尔运算的求值结果。

运算	结果
a b	[01101001] [01010101]
~a ~b	
a & b a | b a ^ b	

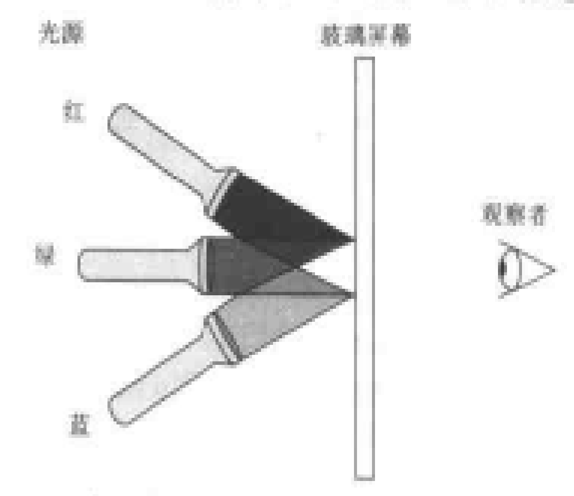
位向量的一个有用的应用就是表示有限集合。例如，我们能够用位向量[a_{w-1}，…，a_1，a_0]来表示任何子集 $A \subseteq \{0, 1, \cdots, w-1\}$，其中 $a_i = 1$ 当且仅当 $i \in A$。例如，（记住我们是把 a_{w-1} 写在左边，而将 a_0 写在右边），我们有 a = [01101001]表示集合 $A = \{0, 3, 5, 6\}$，而 b = [01010101]表示集合 $B = \{0, 2, 4, 6\}$。在这种解释中，布尔运算|和&分别相当于集合的并和交，而~相当于集合的补。比如，运算 a&b 得到位向量[01000001]，而 $A \cap B = \{0, 6\}$。

实际上，对于任何集合 S，结构（$\mathcal{P}(S)$, ∪, ∩, ¯, ∅, S）形成了一个布尔代数，其中 $\mathcal{P}(S)$表示所有

S 的子集的集合，而 ¯ 表示集合的补运算符。也就是说，对于任何集合 A，它的补集就是集合 $\overline{A} = \{a \in S \mid a \notin A\}$。使用位向量运算来表示和操作有限集合的能力，是实践一种深奥的数学原理的结果。

练习题 2.9

通过混合三种不同颜色的光（红色、绿色和蓝色），计算机在视频屏幕或者液晶显示器上产生彩色的画面。设想一种简单的方法，使用三种不同颜色的光，每种光都能打开或关闭，投射到玻璃屏幕上，



那么基于光源 R（红）、G（绿）、B（蓝）的关闭（0）或打开（1），我们就能够创建八种不同的颜色：

R	G	B	颜色	R	G	B	颜色
0	0	0	黑色	1	0	0	红色
0	0	1	蓝色	1	0	1	红紫色
0	1	0	绿色	1	1	0	黄色
0	1	1	蓝绿色	1	1	1	白色

这些颜色的集合形成了一个八元素布尔代数。

A. 一种颜色的补是通过关掉那些打开的颜色光，且打开那些关掉的颜色光而形成的。那么上面列出的八种颜色的补是什么？

B. 对于这种代数，布尔值 0"和 1"对应的颜色是什么？

C. 描述对下列颜色应用布尔运算的结果：

蓝色	|	红色	=
红紫色	&	蓝绿色	=
绿色	^	白色	=

2.1.8 C 中的位级运算

C 的一个很有用的特性就是它支持按位布尔运算。事实上，我们在布尔运算中使用的那些符号就是在 C 中使用的：|就是 OR，&就是 AND，~就是 NOT，而^就是 EXCLUSIVE-OR。这些运算能运用到任何"整型"的数据类型上，也就是，那些声明为 char 或者 int 的数据类型，无论有没有像

short、long 或者 unsigned 这样的限定词。以下是一些表达式求值的例子：

C 的表达式	二进制表达式	二进制结果	C 的结果
~0x41	~[01000001]	[10111110]	0xBE
~0x00	~[00000000]	[11111111]	0xFF
0x69&0x55	[01101001]&[01010101]	[01000001]	0x41
0x69|0x55	[01101001]|[01010101]	[01111101]	0x7D

正如我们的示例说明的，确定一个位级表达式的结果的最好方法就是将十六进制参数扩展成它们的二进制表示，执行二进制运算，然后再转换回十六进制。

练习题 2.10

为了展示^的环属性的用处，考虑下面的程序：

```
1 void inplace_swap(int *x, int *y)
2 {
3       *x = *x ^ *y; /* Step 1 */
4       *y = *x ^ *y; /* Step 2 */
5       *x = *x ^ *y; /* Step 3 */
6 }
```

正如程序名字所暗示的那样，我们认为这个过程的效果是交换指针变量 x 和 y 所指向的存储位置处存放的值。注意，与通常的交换两个数值的技术不一样，当我们移动一个值时，我们不需要第三个位置来临时存储另一个值。这种交换方式并没有性能上的优势，它仅仅是一个智力上的消遣。

开始时，指针 x 和 y 所指向的位置存储的值分别是 a 和 b，填写下表，给出在程序的每一步之后，存储在这两个位置中的值。利用环的属性来表明所希望的效果被达到了。回想一下，每个元素就是它自身的加法逆元（也就是说，a ^ a = 0）。

步骤	*x	*y
初始	a	b
第一步		
第二步		
第三步		

位级运算的一个常见用法就是实现掩码运算，这里掩码是一个位模式，表示从一个字中选出的一组位。让我们来看一个例子，掩码 0xFF（最低有效八位为 1）表示一个字的低位字节。位级运算 x&0xFF 生成一个由 x 的最低有效字节组成的值，而其他的字节就被置为了 0。比如，对于 x=0x89ABCDEF，表达式将得到 0x000000EF。表达式~0 将生成一个全 1 的掩码，不管机器的字大小是多少。尽管对于一个 32 位机器同样的掩码可以写成 0xFFFFFFFF，但是这样的代码是不可移植的。

练习题 2.11

根据下面的值，以及当 x=0x98FDECBA 和字长为 32 位时的结果，写出 C 的表达式，方括号中所示的即为结果。

A. x 的最低有效字节，其他位均置为 1[0xFFFFFFBA]。

B. x 的最低有效字节的补，其余字节保持不变[0x98FDEC45]。

C. 除了 x 的最低有效字节外的所有字节保持不变，最低有效字节置为 0[0x98FDEC00]。

尽管我们的例子假设的是 32 位字长，但是你的代码应该可以工作在 w≥8 的任何字长下。

练习题 2.12

从 20 世纪 70 年代末到 80 年代末，Digital Equipment 的 VAX 计算机是一种非常流行的机型。它没有布尔运算 AND 和 OR 指令，它只有 bis（位设置）和 bic（位清除）这两种指令。两种指令的输入都是一个数据字 x 和一个掩码字 m。它们生成一个结果 z，z 是由根据掩码 m 的位修改 x 的位得到的。使用 bis 指令，这种修改就是在 m 为 1 的每个位置，将 z 设置为 1。使用 bic 指令，这种修改就是在 m 为 0 的每个位置，将 z 设置为 0。

我们想要编写 C 函数 bis 和 bic 来计算这两个指令的效果。使用 C 的位级运算，填写下列代码中缺失的表达式：

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ___________;
  return result;
}
/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit clear */
  int result = ___________;
  return result;
}
```

2.1.9 C 中的逻辑运算

C 还提供了一系列的逻辑运算符||、&&和!，分别对应于命题逻辑中的 OR、AND 和 NOT 运算。逻辑运算很容易和位级运算相混淆，但是它们的功能是完全不同的。逻辑运算认为所有非零的参数都表示 TRUE，而参数零表示 FALSE。它们返回 1 或者 0，分别表示结果为 TRUE 或者为 FALSE。以下是一些表达式求值的示例：

表达式	结果
!0x41	0x00
!0x00	0x01
!!0x41	0x01
0x69&&0x55	0x01
0x69||0x55	0x01

可以观察到，按位运算只有在特殊情况中，也就是参数被限制为 0 或者 1 时，才和与其对应的逻辑运算有相同的行为。

逻辑运算符&&和||与它们对应的位级运算&和|之间第二个重要的区别是，如果对第一个参数求值就能确定表达式的结果，那么逻辑运算符就不会对第二个参数求值。因此，例如，表达式 a&&5/a 将不会造成被零除，而表达式 p&&*p++也不会导致间接引用空指针。

练习题 2.13

假设 x 和 y 的字节值分别为 0x66 和 0x93。填写下表，指明各个 C 表达式的字节值：

表达式	值	表达式	值
x & y		x && y	
x | y		x || y	
~x | ~y		!x || !y	
x & !y		x && ~y	

练习题 2.14

只使用位级和逻辑运算，编写一个 C 表达式，它等价于 x = = y。换句话说，当 x 和 y 相等时它将返回 1，否则就返回 0。

2.1.10 C 中的移位运算

C 还提供了一系列的移位运算，以向左或者向右移动位模式。对于一个位表示为$[x_{n-1}, x_{n-2}, \cdots, x_0]$的运算数 x，C 表达式 x << k 会生成一个值，其位表示为$[x_{n-k-1}, x_{n-k-2}, \cdots, x_0, 0, \cdots, 0]$。也就是说，$x$ 向左移动 k 位，丢弃 k 个最高位，并在右端补了 k 个 0。移位量应该是一个 $0 \sim n-1$ 之间的值。移位运算从左至右结合，所以 x<<j<<k 等价于(x<<j)<<k。注意运算符的优先级：1<<5-1 应该按照 1<<(5-1)而不是(1<<5)-1 来求值。

有一个相应的右移运算 $x >> k$，但是它的行为有点微妙。一般而言，机器支持两种形式的右移：逻辑的和算术的。逻辑右移在左端补 k 个 0，得到的结果是$[0, \cdots, 0, x_{n-1}, x_{n-2}, \cdots, x_k]$。算术右移是在左端补 k 个最高有效位的拷贝，得到的结果是$[x_{n-1}, \cdots, x_{n-1}, x_{n-1}, x_{n-2}, \cdots, x_k]$。这种做法看上去可能有点奇特，但是我们会发现它对有符号整数数据的运算非常有用。

C 标准并没有明确定义应该使用哪种类型的右移。对于无符号数据（也就是，以限定词 unsigned 声明的整型对象），右移必须是逻辑的。而对于有符号数据（默认），算术的或者逻辑的右移都可以。不幸地，这就意味着任何假设一种或者另一种右移形式的代码都潜在地会遇到可移植性问题。然而，实际上，几乎所有的编译器/机器组合都对有符号数据使用算术右移，且许多程序员也都假设使用这种右移。

练习题 2.15

填写下表，展示不同移位运算对单字节数的影响。思考移位运算的最好方式是使用二进制表示方式。将最初的值转换为二进制，执行移位运算，然后再将它转换回十六进制。每个答案都应该是 8 个二进制数字或者 2 个十六进制数字。

x		x<<3		x>>2 (逻辑的)		x>>2 (算术的)	
十六进制	二进制	二进制	十六进制	二进制	十六进制	二进制	十六进制
0xF0							
0x0F							
0xCC							
0x55							

2.2 整数表示

在本节中，我们描述用位来编码整数的两种不同的方式：一种只能表示非负数，而另一种能够表示负数、零和正数。在后面我们将会看到它们在数学属性和机器级实现方面有很强的关联。我们还会研究扩展或者收缩一个已编码整数以适应不同长度表示的效果。

2.2.1 整型数据类型

C 支持多种整型数据类型——表示有限范围的整数。这些类型如图 2.8 所示。每种类型都有一个大小指示符：char、short、int 和 long，同时还有被表示的数字是非负数（声明为 unsigned），或者可能是负数（默认）的指示。图 2.2 中给出了对这些不同的大小的典型分配。如图 2.8 所示，这些不同的大小允许表示不同范围的值。C 标准定义了每种数据类型必须能够表示的最小数值范围。如图中所示，虽然 C 标准允许 16 位的表示，但一个典型的 32 位机器使用一个 32 位表示来表示 int 和 unsigned 数据类型。像图 2.2 所描述的，Compaq Alpha 使用 64 位字来表示 long 整数，无符号数的上限超过了 1.84×10^{19}，而有符号数的范围超过了 $\pm9.22\times10^{18}$。

C 声明	保证的		典型 32 位机器	
	最小值	最大值	最小值	最大值
char	-127	127	-128	127
unsigned char	0	255	0	255
short[int]	-32 767	32 767	-32 768	32 767
unsigned short[int]	0	63 535	0	63 535
int	-32 767	32 767	-2 147 483 648	2 147 483 647
unsigned[int]	0	65 535	0	4 294 967 295
long[int]	-2 147 483 647	2 147 483 647	-2 147 483 648	2 147 483 647
unsigned long[int]	0	4 294 967 295	0	4 294 967 295

图 2.8 C 的整型数据类型

方括号中的文字是可选的。

给 C 语言初学者：C、C++和 Java 中的有符号和无符号数

C 和 C++都支持有符号（默认）和无符号数。Java 只支持有符号数。

2.2.2 无符号和二进制补码编码

假设有一个整数数据类型有 w 位。我们可以将位向量写成 $\vec{x}$，表示整个向量，或者写成$[x_{w-1}, x_{w-2}, \cdots, x_0]$，表示向量中的每一位。把 $\vec{x}$ 看做一个写成二进制表示的数，我们就获得了 $\vec{x}$ 的无符号表示。我们用一个函数 $B2U_w$（代表“无符号的二进制”，长度为 w）来表示这种形式：

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

在这个等式中，符号“$\doteq$”表示左手边被定义为等于右手边。函数 $B2U_w$ 将一个长度为 w 的 0、1 串映射到非负整数。它的最小值是用位向量[00…0]表示，也就是整数值 0，而它的最大值是用位向量[11…1]表示，也就是整数值 $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$。因此，函数 $B2U_w$ 能够被定义为一个映射 $B2U_w$:$\{0, 1\}_w \to \{0, \cdots, 2^w-1\}$。注意 $B2U_w$ 是一个双射——对于每一个长度为 w 的位向量，都有一个惟一的值与之对应；反过来，在 $0\sim 2^w-1$ 之间的每一个整数都有一个惟一的长度为 w 的位向量二进制表示与之对应。

对于许多应用，我们还希望表示负数值。最常见的有符号数的计算机表示方式就是二进制补码（two's-complement）形式。它的定义将字的最高有效位解释为负权（negative weight）。我们用函数 $B2T_w$（表示“二进制到二进制补码”，长度为 w）来表示这种解释：

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

最高有效位也称为符号位（sign bit）。当被设置为 1 时，表示值为负，而当被设置为 0 时，值为非负。它能表示的最小值是位向量[10…0]（也就是，设置这个位为负权，但是清除其他所有的位），其整数值为 $TMin_w \doteq -2^{w-1}$。而最大值是位向量[01…1]，其整数值为 $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$。同样地，我们可以看到 $B2T_w$ 也是一个双射 $B2U_w$:$\{0, 1\}^w \to \{-2^{w-1}, \cdots, 2^{w-1}-1\}$，对于可表示范围内的每个位模式都有一个惟一的整数与之对应。

练习题 2.16

假设 w=4，我们能给每个可能的十六进制数字赋予一个数值，假设是一个无符号或者二进制补码表示。请根据这些表示，通过写出等式 2.1 和 2.2 所示的求和公式中的二的非零幂数，填写下表：

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
十六进制	二进制		
A	[1010]	$2^3+2^1=10$	$-2^3+2^1=-6$
0			
3			
8			
C			
F			

图 2.9 展示了不同字长的几个“有趣的”数字的位模式和数值。前三个给出的是可表示的整数的

范围。有几点值得注意。第一，二进制补码的范围是不对称的：$|TMin_w| = |TMax_w| + 1$，也就是说，$TMin_w$ 没有与之对应的正数。就像我们会看到的，这导致了二进制补码运算的某些特殊的属性，并且容易造成程序中细微的错误。第二，最大的无符号值刚好比二进制补码的最大值的两倍大一点：$UMax_w = 2TMax_w + 1$。这是因为二进制补码表示保留了一半的位模式来表示负数值。其他的情况是常数-1 和 0，注意-1 和 $UMax_w$ 有同样的位表示——一个全 1 的串。数值 0 在两种表示方式中都是全 0 的串。

数	字长 w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFFFFFFFFFF 9 223 372 036 854 775 807
$TMin_w$	0x80 -128	0x8000 -32 768	0x80000000 -2 147 483 648	0x8000000000000000 9 223 372 036 854 775 808
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

图 2.9 "有趣的"数字

给出了数字值和十六进制表示。

C 的标准并没有要求要用二进制补码形式来表示有符号整数，但是几乎所有的机器都是这么做的。为了保证代码的可移植性，除了图 2.2 所示的那些范围之外，我们不应该假设任何可表示的数值范围，或者假设它们会被如何表示。C 库中的文件<limits.h>定义了一组常量，来限定运行编译器的这台机器的不同整型数据类型的范围。比如，它定义了常量 INT_MAX、INT_MIN 和 UINT_MAX，它们描述了有符号和无符号整数的范围。对于一个二进制补码的机器，数据类型 int 有 w 位，这些常量就对应于 $TMax_w$、$TMin_w$ 和 $UMax_w$。

旁注：有符号数的其他表示方法

有符号数另外还有两种标准的表示方法：

二进制反码（Ones' Complement，又译作"一的补码"）：这和二进制补码是一致的，除了最高有效位的权是$-(2^{w-1}-1)$而不是-2^{w-1}：

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1}-1)+\sum_{i=0}^{w-2}x_i 2^i$$

符号数值（Sign-Magnitude）：最高有效位是符号位，确定剩下的位应该取负权还是正权：

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2}x_i 2^i\right)$$

这两种表示方法都有一个古怪的属性，那就是对于数字 0 有两种不同的编码方式。对于两种表示方法，[00…0]都被解释为+0。而值-0 在符号量形式中表示为[10…0]，而在二进制反码中表示为[11…1]。虽然过去生产过基于二进制反码表示的机器，但是几乎所有的现代机器都使用二进制补码。我们将看到符号数值编码方式使用在浮点数中。

请注意二进制补码（Two's-complement）和二进制反码（Ones' complement）中撇号的位置是不同的。

作为一个示例，考虑下面的代码：

```
1 short int x = 12345;
2 short int mx = -x;
3
4 show_bytes((byte_pointer) &x, sizeof(short int));
5 show_bytes((byte_pointer) &mx, sizeof(short int));
```

权	12 345		-12 345		53 191	
	位	值	位	值	位	值
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1 024	0	0	1	1 024	1	1 024
2 048	0	0	1	2 048	1	2 048
4 096	1	4096	0	0	0	0
8 192	1	8192	0	0	0	0
16 384	0	0	1	16 384	1	16 384
±32 768	0	0	1	-32 768	1	32 768
总计		12 345		-12 345		53 191

图 2.10 12 345 和-12 345 的二进制补码表示，以及 53 191 的无符号表示

注意后面两个数有相同的位表示。

当运行在大端法机器上时，这段代码的输出为 30 39 和 cf c7，指明 x 的十六进制表示为 0x3039，而 mx 的十六进制表示为 0xCFC7。将它们展开为二进制，我们得到 x 的位模式为[0011000000111001]，而 mx 的位模式为[1100111111000111]。如图 2.10 所示，等式 2.2 对这两个位模式生成的值为 12 345 和-12 345。

练习题 2.17

在第 3 章中，我们将看到由反汇编器生成的列表，反汇编器是一种将可执行程序文件转换回更可读的 ASCII 码形式的程序。这些文件包含许多十六进制数字，典型地都是用二进制补码形式来表示这些值的。能够认识这些数字并理解它们的意义（例如，它们是正数还是负数），是一项重要的技巧。

在下面的列表中，对于标号为A~K（标记在右边）的那些行，将指令名（sub、push、mov和add）右边显示的十六进制值转换为它们的十进制等价值。

```
80483b7: 81 ec 84 01 00 00  sub    $0x184,%esp                 A.
80483bd: 53                 push   %ebx
80483be: 8b 55 08           mov    0x8(%ebp),%edx              B.
80483c1: 8b 5d 0c           mov    0xc(%ebp),%ebx              C.
80483c4: 8b 4d 10           mov    0x10(%ebp),%ecx             D.
80483c7: 8b 85 94 fe ff ff  mov    0xfffffe94(%ebp),%eax       E.
80483cd: 01 cb              add    %ecx,%ebx
80483cf: 03 42 10           add    0x10(%edx),%eax             F.
80483d2: 89 85 a0 fe ff ff  mov    %eax,0xfffffea0(%ebp)       G.
80483d8: 8b 85 10 ff ff ff  mov    0xffffff10(%ebp),%eax       H.
80483de: 89 42 1c           mov    %eax,0x1c(%edx)             I.
80483e1: 89 9d 7c ff ff ff  mov    %ebx,0xffffff7c(%ebp)       J.
80483e7: 8b 42 18           mov    0x18(%edx),%eax             K.
```

2.2.3 有符号数和无符号数之间的转换

既然 $B2U_w$ 和 $B2T_w$ 都是双射，它们就有定义明确的逆映射。将 $U2B_w$ 定义为 $B2U_w^{-1}$，而将 $T2B_w$ 定义为 $B2T_2^{-1}$。这些函数给出了一个数值的无符号或者二进制补码的位模式。给定 $0 \le x < 2^w$ 范围内的一个整数 x，函数 $U2B_w(x)$会给出 x 的惟一的 w 位无符号表示。相似地，当 x 满足 $-2^{w-1} \le x < 2^{w-1}$，函数 $T2B_w(x)$会给出 x 的惟一的 w 位二进制补码表示。可观察到，对于范围 $0 \le x < 2^{w-1}$ 内的值，这两个函数将生成同样的位模式——最高位是0，因此这个位是正权还是负权就没有关系了。

考虑函数 $U2T_w(x) \doteq B2T_w(U2B_w(x))$，其输入是一个 $0 \sim 2^w-1$ 之间的值，得到一个 $-2^{w-1} \sim 2^{w-1}-1$ 之间的值，这里两个数有相同的位模式，除了参数是无符号的，而结果是以二进制补码表示的。相反地，函数 $T2U_w(x) \doteq B2U_w(T2B_w(x))$ 生成一个无符号数，它和 x 的二进制补码值有相同的位表示。例如，如图2.10所示，-12 345的16位二进制补码表示就和53 191的16位无符号表示相同。因此，$T2U_{16}(-12\ 345) = 53\ 191$，并且 $U2T_{16}(53\ 191) = -12\ 345$。

这两个函数看上去好象只有理论价值，但实际上它们有非常大的实际意义——它们形式化地定义了C中有符号和无符号值之间的强制类型转换的结果。例如，设想在一台二进制补码机器上执行下列代码：

```
1 int x = -1;
2 unsigned ux = (unsigned) x;
```

因为从图2.9中我们可以看到-1的 w 位二进制补码表示和 $UMax_w$ 有相同的位表示，所以这段代码将把ux设置为 $UMax_w$，其中 w 是数据类型int中的位数。一般而言，从一个有符号值x强制类型转换到无符号数值（unsigned）x就相当于应用函数 $T2U$。强制类型转换并没有改变参数的位表示，只是改变了如何将这些位解释为一个数字。相似地，从无符号值 u 强制类型转换到有符号值(int)u就相当于应用函数 $U2T$。

练习题 2.18

利用你解答练习题2.16时填写的表格，填写下列描述函数 $T2U_4$ 的表格：

x	$T2U_4(x)$
−8	
−6	
−4	
−1	
0	
3	

为了更好地理解一个有符号数字 x 和与之对应的无符号数 $T2U_w(x)$，我们可以利用它们有相同的位表示这一事实来推导出一个数字关系。比较等式 2.1 和 2.2，我们可以发现对于位模式 $\vec{x}$，如果我们计算 $B2U_w(\vec{x}) - B2T_w(\vec{x})$ 之差，从 0 到 $w-2$ 的位的加权和将互相抵消掉，剩下一个值：$B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$。这就得到一个关系：$B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$。如果我们让 $x = B2T_w(\vec{x})$，我们就有

$$B2U_w(T2B_w(x)) = T2U_w(x) = x_{w-1}2^w + x \quad (2.3)$$

这个关系对于证明无符号和二进制补码运算之间的关系是很有用的。在 x 的二进制补码表示中，位 x_{w-1} 决定了 x 是否为负，得到

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad (2.4)$$

图 2.11 说明了函数 $T2U$ 的行为。就像它所说明的，当将一个有符号数映射为它相应的无符号数时，负数就被转换成了大的正数，而非负数会保持不变。

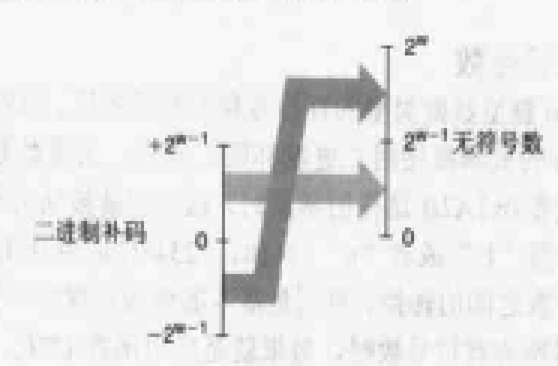


图 2.11 从二进制补码到无符号数的转换

函数 $T2U$ 将负数转换为大的正数。

练习题 2.19

请说明等式 2.4 是如何应用到你在解答练习题 2.18 时生成的表格中的各项的。

反过来看，我们希望推导出一个无符号数 x 和与之对应的有符号数 $U2T_w(x)$ 之间的关系。如果我们设 $x = B2U_w(\vec{x})$，我们有

$$B2T_w(U2B_w(x)) = U2T_w(x) = -x_{w-1}2^w + x \quad (2.5)$$

在 x 的无符号表示中，位 x_{w-1} 决定了 x 是否大于或者等于 2^{w-1}，得到

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \quad (2.6)$$

图 2.12 说明了这个行为。对于小的数（$<2^{w-1}$），从无符号到有符号的转换将保留数字的原值。对于大的数（$\geq 2^{w-1}$），数字将被转换为一个负数值。

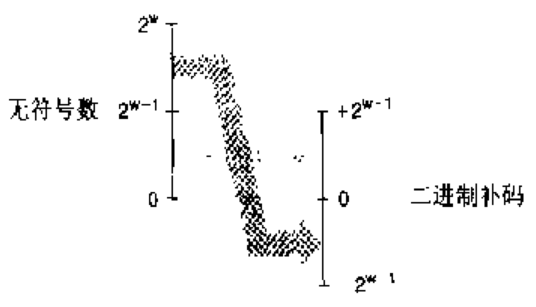


图 2.12 从无符号数到二进制补码的转换

函数 $U2T$ 把大于 $2^{w-1}-1$ 的数字转换为负值。

为了总结一下，我们可以考虑无符号与二进制补码表示之间互相转换的结果。对于在范围 $0 \leq x < 2^{w-1}$ 之内的值而言，我们得到 $T2U_w(x) = x$ 和 $U2T_w(x) = x$。也就是说，在这个范围内的数字有相同的无符号和二进制补码表示。对于这个范围以外的数值，转换需要加上或者减去 2^w。例如，我们有 $T2U_w(-1) = -1 + 2^w = UMax^w$——最靠近 0 的负数映射为最大的无符号数。在另一个极端，我们可以看到 $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$——最小的负数映射为一个刚好在二进制补码的正数范围之外的无符号数。使用图 2.10 的示例，我们能看到 $T2U_{16}(-12\,345) = 65\,563 + -12\,345 = 53\,191$。

2.2.4 C 中的有符号与无符号数

如图 2.8 所示，C 支持所有整型数据类型的有符号和无符号运算。尽管 C 标准没有指定某种有符号数的表示，但是几乎所有的机器都使用二进制补码。通常，大多数数字默认都是有符号的。例如，当声明一个像 12345 或者 0x1A2B 这样的常量时，这个值就被认为是有符号的。要创建一个无符号常量，必须加上后缀字符“U”或者“u”（例如，12345U 或者 0x1A2Bu）。

C 允许无符号数和有符号数之间的转换。原则是基本的位表示保持不变。因此，在一台二进制补码机器上，当从无符号数转换为有符号数时，效果就是应用函数 $U2T_w$，而从有符号数转换为无符号数时，就是应用函数 $T2U_w$，其中 w 表示数据类型的位数。

显式的强制类型转换将导致转换的发生，就像下面的代码：

```
1  int tx, ty;
2  unsigned ux, uy;
3
4  tx = (int) ux;
5  uy = (unsigned) ty;
```

另外，当一种类型的表达式被赋值给另外一种类型的变量时，转换是隐式发生的，就像下面的代码：

```
1  int tx, ty;
2  unsigned ux, uy;
```

```
3
4  tx = ux;  /* Cast to signed */
5  uy = ty;  /* Cast to unsigned */
```

当用 printf 来输出数值时，指示符%d、%u 和%x 分别用来以有符号十进制、无符号十进制和十六进制格式来输出一个数字。注意 printf 没有使用任何类型信息，所以它可以用指示符%u 来输出类型 int 的数值，也可以用指示符%d 输出类型 unsigned 的数值。例如，考虑下面的代码：

```
1  int x = -1;
2  unsigned u = 2147483648; /* 2 to the 31st */
3
4  printf("x = %u = %d\n", x, x);
5  printf("u = %u = %d\n", u, u);
```

当在一个 32 位机器上运行时，它的输出如下：

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

在这两个示例中，printf 首先将这个字作为一个无符号数输出，然后把它当作一个有符号数输出。我们可以看看实际运行中的转换函数：$T2U_{32}(-1) = UMax_{32} = 4\,294\,967\,295$ 和 $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$。

由于 C 对同时包含有符号和无符号数的表达式的处理方式，出现了一些奇特的行为。当执行一个运算时，如果它的一个运算数是有符号的而另一个是无符号的，那么 C 会隐含地将有符号参数强制类型转换为无符号数，并假设这两个数都是非负的，来执行这个运算。就像我们会看到的，这种方法对于标准的算术运算来说并无多大差异，但是对于像<和>这样的关系运算符来说，它会导致与直觉不相符的结果。图 2.13 展示了一些关系表达式的示例以及它们得到的求值结果，这里假设使用的是一台 32 位机器和二进制补码表示。与直觉不相符的情况用“*”标出来了。考虑一下比较式-1<0U。因为第二个运算数是无符号的，所以第一个运算数就会隐含地转换为无符号数，因此表达式就等价于4294967295U<0U（回想一下 $T2U_w(-1) = UMax_w$），这个答案显然是错的。另外那些示例也可以通过相似的分析来理解。

表 达 式	类 型	求 值
`0 == 0U`	无符号	1
`-1 < 0`	有符号	1
`-1 < 0U`	无符号	0*
`2147483647 > -2147483647-1`	有符号	1
`2147483647U > -2147483647-1`	无符号	0*
`2147483647 > (int) 2147483648U`	有符号	1*
`-1 > -2`	有符号	1
`(unsigned) -1 > -2`	无符号	1

图 2.13 32 位机器上 C 的升级规则（promotion rule）的效果

非直观的情况被标注了“*”。当比较表达式中的任一运算数是无符号的时候，另一个运算数也被隐式强制转换为无符号。在 C 中，为避免溢出问题，我们必须把 TMin_32 写为-2147483647-1，而不是-2147483648。编译器处理一个形为-X 的表达式的方法是首先读表达式 X，然后对它取反，但是 21474836478 太大了，不能表示为一个 32 位的、二进制补码的数。

练习题 2.20

假设在采用二进制补码运算的32位机器上对这些表达式求值，按照图2.13的风格，填写下表，描述强制类型转换和关系运算的结果：

表达式	类型	求值
`-2147483648 == 2147483648U`		
`-2147483648 < -2147483647`		
`(unsigned) -2147483648 < -2147483647`		
`-2147483648 < 2147483647`		
`(unsigned) -2147483648 < 2147483647`		

2.2.5 扩展一个数字的位表示

一个常见的运算是在不同字长的整数之间转换，同时又保持数值不变。当然，当目标数据类型太小了，以至于不能表示想要的值时，这可能根本就是不可能的。然而，从一个较小的数据类型转换到一个较大的类型，应该总是可能的。要将一个无符号数转换为一个更大的数据类型，我们只要简单地在表示的开头添加0。这种运算被称为零扩展（zero extension）。要将一个二进制补码数字转换为一个更大的数据类型，规则是执行一个符号扩展（sign extension），在表示中添加最高有效位的值。因此，如果我们原始值的位表示为$[x_{w-1}, x_{w-2}, \cdots, x_0]$，那么扩展后的表示就为$[x_{w-1}, \cdots, x_{w-1}, x_{w-1}, x_{w-2}, \cdots, x_0]$。

例如，考虑下面的代码：

```
1  short sx = val;                 /* -12345 */
2  unsigned short usx = sx;        /* 53191 */
3  int x = sx;                     /* -12345 */
4  unsigned ux = usx;              /* 53191 */
5
6  printf("sx = %d:\t", sx);
7  show_bytes((byte_pointer) &sx, sizeof(short));
8  printf("usx = %u:\t", usx);
9  show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf("x = %d:\t", x);
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

在使用二进制补码表示的32位大端法机器上运行这段代码时，将会打印出如下输出：

```
sx  = -12345:   cf c7
usx = 53191:    cf c7
x   = -12345:   ff ff cf c7
ux  = 53191:    00 00 cf c7
```

我们看到，尽管-12 345 的二进制补码表示和 53 191 的无符号表示在 16 位字长时是相同的，但是在 32 位字长时却是不同的。特别地，-12 345 的十六进制表示为 0xFFFFCFC7，而 53 191 的十六进制表示为 0x0000CFC7。前者使用的是符号扩展——16 个最高有效位 1，表示为十六进制就是 0xFFFF，被加作于头的位。后者使用 16 个 0 来扩展，表示为十六进制就是 0x0000。

我们如何证明符号扩展工作得正确呢？我们想要证明的是 $B2T_{w+k}([x_{w-1},\cdots,x_{w-1},x_{w-1},x_{w-2},\cdots,x_0]) = B2T_w([x_{w-1}, x_{w-2}, \cdots, x_0])$，这里，在左手边的表达式中，我们增加了 k 个位 x_{w-1} 的副本。证明是对 k 进行归纳。也就是说，如果我们能够证明符号扩展一位保持了数值不变，那么符号扩展任意位都能保持这种属性。因此，证明的任务就变为了：

$$B2T_{w+1}([x_{w-1}, \cdots, x_{w-1}, x_{w-1}, x_{w-2}, \cdots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \cdots, x_0])$$

用等式 2.2 展开左手边的表达式，会得到：

$$\begin{aligned}
B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2},\cdots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\
&= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= B2T_w([x_{w-1}, x_{w-2},\cdots, x_0])
\end{aligned}$$

我们使用的关键属性是$-2^w + 2^{w-1} = -2^{w-1}$。因此，加上一个权值为$-2^w$的位和将一个权值为$-2^{w-1}$的位转换为一个权值为 2^{w-1}的位，两项运算的综合效果就会保持原始的数值。

值得一提的是，从一个数据大小到另一个数据大小，以及无符号和有符号数字之间的转换的相对顺序能够影响一个程序的行为。考虑我们前面那个例子的如下额外代码：

```
1 unsigned uy = x; /* Mystery! */
2
3 printf("uy = %u:\t", uy);
4 show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

这部分代码产生如下输出：

```
uy = 4294954951: ff ff cf c7
```

这表明表达式：

```
(unsigned) (int) sx            /* 4294954951 */
```

和

```
(unsigned) (unsigned short) sx     /* 53191 */
```

产生了不同的数值，即使原始的和最后的数据类型是相同的。在前一个表达式中，我们首先将 16 位的 short 符号扩展为 32 位的 int，而在后一个表达式中执行的则是零扩展。

练习题 2.21

考虑下面的C函数：

```
int fun1(unsigned word)
{
    return (int) ((word << 24) >> 24);
}
int fun2(unsigned word)
{
    return ((int) word << 24) >> 24;
}
```

假设在一个使用二进制补码运算的32位字长的机器上执行这些函数。还假设有符号数值的右移是算术右移，而无符号数值的右移是逻辑右移。

A. 填写下表，说明这些函数对几个示例参数的结果：

w	fun1(w)	fun2(w)
127		
128		
255		
256		

B. 用语言来描述这些函数执行的有用的计算。

2.2.6 截断数字

假设不用额外的位来扩展一个数值，我们会减少表示一个数字的位数。例如在下面的代码中就发生了这种情况：

```
1  int    x = 53191;
2  short sx = (short) x;     /* -12345 */
3  int    y = sx;            /* -12345 */
```

在一台典型的32位机器上，当我们把 x 强制类型转换为 short 时，我们就将32位的 int 截断为了16位的 short int。就像我们前面所看到的，这个16位的模式就是-12 345的二进制补码表示。当我们把它强制类型转换回 int 时，符号扩展把高16位设置为1，从而生成-12 345的32位二进制补码表示。

当将一个 w 位的数 $\vec{x}=[x_{w-1}, x_{w-2},\cdots, x_0]$ 被截断为一个 k 位数字时，我们会丢弃高 $w-k$ 位，得到一个位向量 $\vec{x}'=[x_{k-1}, x_{k-2},\cdots, x_0]$。截断一个数字可能会改变它的值——溢出的一种形式。我们现在来研究一下什么数值将产生这种情况。对于一个无符号数字 x，截断它到 k 位的结果就相当于计算 x mod 2^k。通过应用模运算到等式2.1就可以看到：

$$
\begin{aligned}
B2U_w([x_w, x_{w-1}, \cdots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\
&= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\
&= \sum_{i=0}^{k-1} x_i 2^i \\
&= B2U_k([x_k, x_{k-1}, \cdots, x_0])
\end{aligned}
$$

在这段推导中，我们利用了属性：对于任何 $i \geq k$，$2^i \bmod 2^k = 0$ 和 $\sum_{i=0}^{k-1} x_i 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$。

对于一个二进制补码数字 x，相似的推理表明 $B2T_w([x_w, x_{w-1}, \cdots, x_0]) \bmod 2^k = B2U_k[x_k, x_{k-1}, \cdots, x_0]$。也就是，$x \bmod 2^k$ 能够被一个位级表示为$[x_{k-1}, \cdots, x_0]$的无符号数表示。不过，一般而言，我们将截断的数字视为有符号的。这将得到数值 $U2T_k(x \bmod 2^k)$。

总而言之，截断的结果如下所示：

$$B2U_k\ [x_k, x_{k-1}, \cdots, x_0] = B2U_w\ ([x_w, x_{w-1}, \cdots, x_0]) \bmod 2^k \tag{2.7}$$

$$B2T_k\ [x_k, x_{k-1}, \cdots, x_0] = U2T_k\ (B2T_w\ ([x_w, x_{w-1}, \cdots, x_0]) \bmod 2^k) \tag{2.8}$$

练习题 2.22

假设我们将一个四位数值（用十六进制数字 0~F 表示）截断到一个三位数值（用十六进制数字 0~7 表示），填写下表，根据那些位模式的无符号和二进制补码表示，说明这种截断对某些情况的结果：

十六进制		无符号		二进制补码	
原始值	截断值	原始值	截断值	原始值	截断值
0	0	0		0	
3	3	3		3	
8	0	8		-8	
A	2	10		-6	
F	7	15		-1	

解释等式 2.7 和 2.8 是如何应用到这些示例上的。

2.2.7 关于有符号数与无符号数的建议

就像我们看到的那样，有符号数到无符号数的隐式强制类型转换导致了某些与直觉不相符的行为。而这些与直觉不相符的特性经常导致程序错误，并且包含隐式强制类型转换的细微差别的错误很难被发现。因为这种强制类型转换是看不到的，我们经常忽视了它的影响。

练习题 2.23

考虑下列代码，这段代码试图计算数组 a 中所有元素的和，其中元素的数量由参数 length 给出：

```
1 /* WARNING: This is buggy code */
2 float sum_elements(float a[], unsigned length)
3 {
4     int i;
5     float result = 0;
6
7     for (i = 0; i <= length-1; i++)
8 result += a[i];
9     return result;
10 }
```

当运行时参数 length 等于零，这段代码应该返回 0.0。但实际上，代码会遇到存储器（memory）错误。请解释为什么会发生这样的情况，并且说明该如何修改代码。

避免这类错误的一种方法就是绝不使用无符号数。实际上，除了 C 以外很少有语言支持无符号整数。很明显，这些其他语言的设计者认为它们的麻烦要比益处多得多。比如，Java 只支持有符号整数，并且要求以二进制补码运算来实现。正常的右移运算符>>被定义为执行算术右移。特殊的运算符>>>被指定为执行逻辑右移。

当我们想要把字仅仅看做是位的集合而没有任何数字意义时，无符号数值是非常有用的。例如，往一个字中放入描述各种布尔条件的标记（flag）时，就是这样。地址自然是无符号的，所以系统程序员发现无符号类型是很有帮助的。当实现模运算和多精度运算的数学包时，数字是由字的数组来表示的，无符号值也会非常有用。

2.3 整数运算

许多刚入门的程序员非常惊奇地发现，两个正数相加会得出一个负数，而且比较表达式 $x<y$ 和比较表达式 $x-y<0$ 会产生不同的结果。这些属性是计算机运算的有限性造成的。理解计算机运算的细微之处能够帮助程序员编写更可靠的代码。

2.3.1 无符号加法

考虑两个非负整数 x 和 y，满足 $0 \le x, y \le 2^w-1$。每个数都能表示为 w 位无符号数字。然而，如果我们计算它们的和，我们就有一个可能的范围 $0 \le x+y \le 2^{w+1}-2$。表示这个和可能需要 $w+1$ 位。例如，图 2.14 展示了当 x 和 y 有四位表示时，函数 $x+y$ 的坐标图。参数（显示在水平轴上）取值范围为 0～15，但是和的取值范围为 0～30。函数的形状是一个有坡度的平面。如果我们保持和为一个 $w+1$ 位的数字，并且把它加上另外一个数值，我们可能需要 $w+2$ 个位，以此类推。这种持续的“字长膨胀”意味着，要想完整地表示算术运算的结果，我们不能对字长做任何限制。一些编程语言，例如 Lisp，实际上就支持无限精度的运算，允许任意的（当然，要在机器的存储器限制之内）整数运算。更常见的是，编程语言支持固定精度的运算，因此像“加法”和“乘法”这样的运算不同于它们在整数上的相应运算。

无符号运算可以被视为一种形式的模运算。无符号加法等价于计算模 2^w 的和。可以通过简单地丢弃 $x+y$ 的 $w+1$ 位表示的高位，来计算这个数值。比如，考虑一个四位数字表示，$x=9$ 和 $y=12$ 的位表示分别为[1001]和[1100]。它们的和是 21，五位的表示为[10101]。但是如果我们丢弃最高位，

我们就得到[0101]，也就是说，十进制值的 5。这就和值 21 mod 16 = 5 一致。

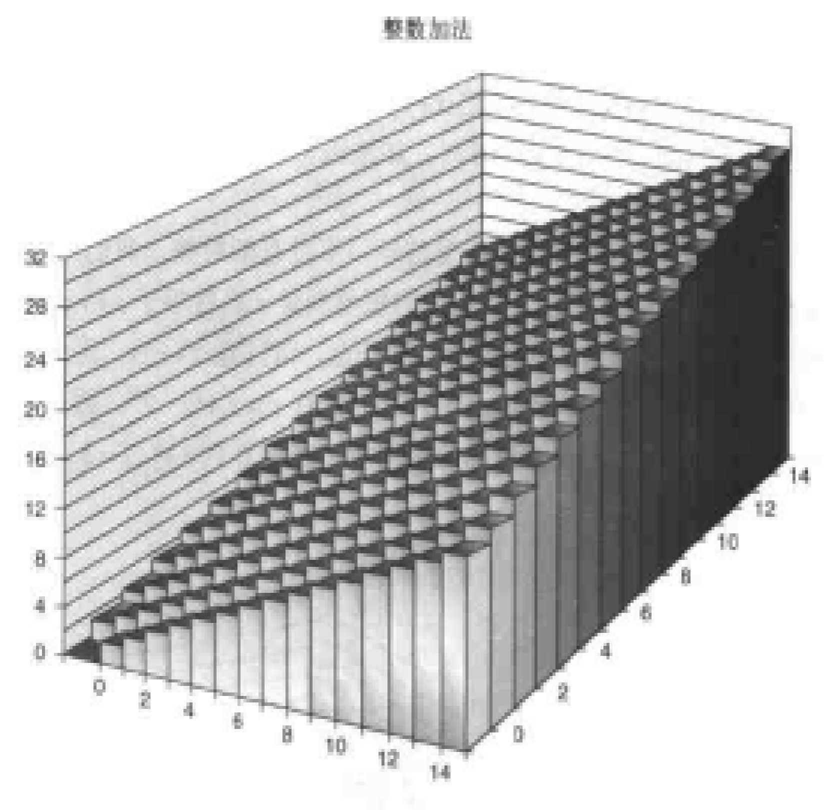


图 2.14 整数加法

对于一个四位的字长，其和可能需要五位。

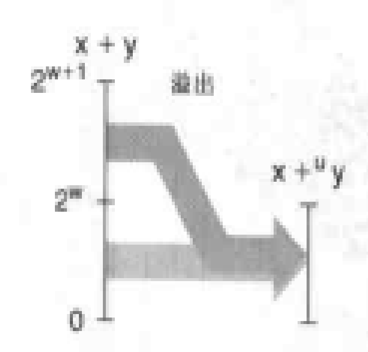


图 2.15 整数加法和无符号加法间的关系

当 $x+y$ 大于 2^w-1 时，其和将溢出。

一般而言，我们可以看到，如果 $x+y<2^w$，和的 $w+1$ 位表示中的最高位会等于 0，因此丢弃它不会改变这个数值。另一方面，如果 $2^w \le x+y<2^{w+1}$，和的 $w+1$ 位表示中的最高位会等于 1，因此丢弃它就相当于从和中减去了 2^w。图 2.15 说明了这两种情况。这会得到一个范围 $0 \le x+y-2^w < 2^{w+1}-2^w = 2^w$ 中的值，刚好等于 x 与 y 的和，然后模 2^w 的结果。让我们来定义参数 x 和 y 的运算 $+^u_w$，这里 $0 \le x, y < 2^w$，如下：

$$x +^u_w y = \begin{cases} x+y, & x+y<2^w \\ x+y-2^w, & 2^w \le x+y < 2^{w+1} \end{cases} \tag{2.9}$$

这正好是在 C 中执行两个 w 位无符号数值加法时我们得到的结果。

说一个算术运算溢出了，是指完整的整数结果不能放到数据类型的字长限制中去。如等式 2.9 所示，当两个运算数的和为 2^w 或者更大时，就发生了溢出。图 2.16 展示了字长为 4 的无符号加法函数的坐标图。这个和是按模 $2^4=16$ 计算的。当 $x+y<16$ 时，没有溢出，并且 $x+^u_4 y$ 就是 $x+y$。这对应于图中标记为"正常"的斜面。当 $x+y \geq 16$ 时，加法溢出，结果相当于从和中减去 16。这对应于图中标记为"溢出"的斜面。

当执行 C 程序时，不会将溢出作为错误而发信号。不过有的时候，我们可能希望判定是否发生了溢出。比如，假设我们计算 $s \doteq x+^u_w y$，并且我们想要判定 s 是否等于 $x+y$。我们声称当且仅当 $s<x$（或者等价地 $s<y$）时，发生了溢出。要弄明白这一点，请注意 $x+y \geq x$，因此如果 s 没有溢出，我们能够肯定 $s \geq x$。另一方面，如果 s 确实溢出了，我们就有 $s=x+y-2^w$。假设 $y<2^w$，我们就有 $y<2^w<0$，因此 $s=x+y-2^w<x$。在我们前面的示例中，我们看到 $9+^u_4 12=5$。既然 $5<9$，我们就可以看出发生了溢出。

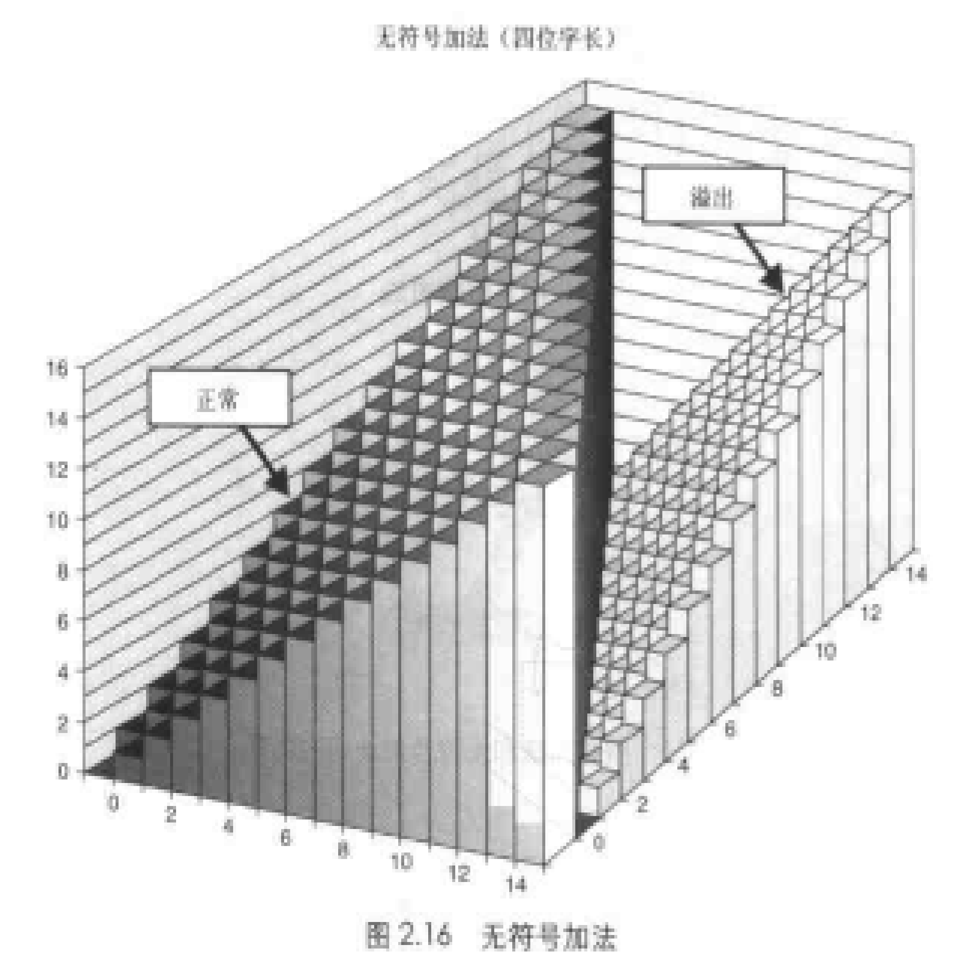


图 2.16 无符号加法

使用四位字长，加法是模 16 的。

模数加法形成了一种数学结构，称为阿贝尔群（Abelian group），这是以丹麦数学家 Niel Henrik Abel（1802～1829）的名字命名的。也就说，它是可交换的（这就是为什么叫"Abelian"的地方）和可结合的。它有一个单位元 0，并且每个元素有一个加法逆元。让我们考虑这样一种情况：w 位的无符号数的集合，执行加法运算 $+^u_w$。对于每个值 x，必然有某个值 $-^u_w x$ 满足 $-^u_w x+^u_w x=0$。当

$x=0$ 时，加法逆元显然是 0。对于 $x>0$，考虑值 2^w-x。我们观察到这个数字在 $0\le 2^w-x<2^w$ 范围之内，并且 $(x+2^w-x) \bmod 2^w = 2^w \bmod 2^w = 0$。因此，它就是 x 在 $+^u_w$ 下的逆元。这两种情况就导出了对于 $0\le x<2^w$ 的等式：

$$-^u_w x = \begin{cases} x, & x=0 \\ 2^w-x, & x>0 \end{cases} \qquad (2.10)$$

练习题 2.24

我们能用一个十六进制数字来表示长度 $w=4$ 的位模式。对于这些数字的无符号解释，使用等式 2.10 填写下表，给出所示数字的无符号加法逆元的位表示（用十六进制形式）。

x		$-^u_4 x$	
十六进制	二进制	二进制	十六进制
0			
3			
8			
A			
F			

2.3.2 二进制补码加法

对于二进制补码加法也有类似的问题。给定在范围 $-2^{w-1}\le x, y\le 2^{w-1}-1$ 之内的整数值 x 和 y，它们的和就在范围 $-2^w\le x+y\le 2^w-2$ 之内，可能需要 $w+1$ 位来表示。就像以前一样，我们通过将表示截断到 w 位，来避免数据大小的扩张。然而，结果在数学上却不像模数加法那样。

两个数的 w 位二进制补码之和与无符号之和有完全相同的位级表示。实际上，大多数计算机使用同样的机器指令来执行无符号或者有符号加法。因此，我们能够定义字长为 w 的二进制补码加法，在运算数 x 和 y 上表示为的 $+^t_w$，满足 $-2^{w-1}\le x, y\le 2^{w-1}$：

$$x +^t_w y \doteq U2T_w(T2U_w(x) +^u_w T2U_w(y)) \qquad (2.11)$$

根据等式 2.3，我们可以把 $T2U_w(x)$ 写成 $x_{w-1}2^w+x$，把 $T2U_w(y)$ 写成 $y_{w-1}2^w+y$。使用属性，即 $+^u_w$ 是模 2^w 的加法，以及模数加法的属性，我们就能得到：

$$\begin{aligned} x +^t_w y &= U2T_w(T2U_w(x) +^u_w T2U_w(y)) \\ &= U2T_w[(-x_{w-1}2^w + x + -y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x+y) \bmod 2^w] \end{aligned}$$

消除了 $x_{w-1}2^w$ 和 $y_{w-1}2^w$ 这两项，因为它们模 2^w 等于 0。

为了更好地理解这个数量，让我们定义 z 为整数和 $z\doteq x+y$，z' 为 $z'\doteq z \bmod 2^w$，而 z'' 为 $z''=U2T_w(z')$。数值 z'' 等于 $x +^t_w y$。我们把分析分解成四种情况，如图 2.17 所示。

1. $-2^w\le z<-2^{w-1}$。然后，我们将有 $z'=z+2^w$。这就得出 $0\le z'<-2^{w-1}+2^w=2^{w-1}$。检查等式 2.6，我

们看到 z' 在满足 $z''=z'$ 的范围之内。这种情况被称为负溢出（negative overflow）。我们将两个负数 x 和 y 相加（这是我们能得到 $z<-2^{w-1}$ 的惟一方式），得到一个非负的结果 $z''=x+y+2^w$。

2. $-2^{w-1}\le z<0$。那么，我们又将有 $z'=z+2^w$，得到 $-2^{w-1}+2^w=2^{w-1}\le z'<2^w$。检查等式 2.6，我们看到 z' 在满足 $z''=z'-2^w$ 的范围之内，因此 $z''=z'-2^w=z+2^w-2^w=z$。也就是说，我们的二进制补码和 z'' 等于整数和 $x+y$。

3. $0\le z<2^{w-1}$。那么，我们将有 $z'=z$，得到 $0\le z'<2^{w-1}$，因此 $z''=z'=z$。二进制补码和 z'' 又等于整数和 $x+y$。

4. $2^{w-1}\le z<2^w$。那么，我们又将有 $z'=z$，得到 $2^{w-1}\le z'<2^w$。但是在这个范围内，我们有 $z''=z'-2^w$，得到 $z''=x+y-2^w$。这种情况被称为正溢出（positive overflow）。我们将正数 x 和 y 相加（也是我们能得到 $z\ge 2^{w-1}$ 的惟一方式），得到一个负数结果 $z''=x+y-2^w$。

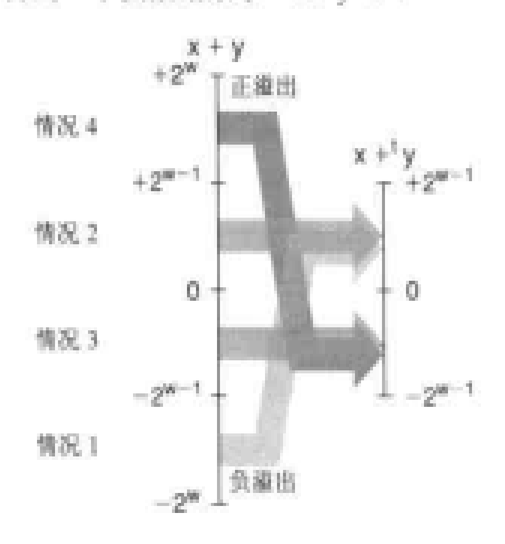


图 2.17 整数和二进制补码加法之间的关系

当 $x+y$ 小于 -2^{w-1} 时，产生负溢出。当它大于 $2^{w-1}+1$ 时，产生正溢出。

通过前面的分析，我们可以给出当对在范围 $-2^{w-1}\le x, y\le 2^{w-1}-1$ 之内的 x 和 y 实施运算 $+^t_w$ 时，我们有下面这样的式子：

$$x+^t_w y=\begin{cases}x+y-2^w, & 2^{w-1}\le x+y \quad \text{正溢出}\\ x+y, & -2^{w-1}\le x+y<2^{w-1} \quad \text{正常}\\ x+y+2^w, & x+y\le -2^{w-1} \quad \text{负溢出}\end{cases} \tag{2.12}$$

作为说明，图 2.18 展示了一些四位二进制补码加法的示例。每个示例的情况都被标号为对应于等式 2.12 的推导过程中的情况。注意 $2^4=16$，因此负溢出得到的结果比整数和大 16，而正溢出得到的结果比之小 16。我们包括了运算数和结果的位级表示。我们可以观察到，能够通过对运算数执行二进制加法并将结果截断到四位，从而得到结果。

图 2.19 阐述了字长 $w=4$ 的二进制补码加法。运算数的范围为 -8～7 之间。当 $x+y<-8$ 时，二进制补码加法就会负溢出，导致和增加了 16。当 $-8\le x+y<8$ 时，加法就产生 $x+y$。当 $x+y\ge 8$，加法就会正溢出，使得和减少了 16。这三种情况中的每一种都形成了图中的一个斜面。

等式 2.12 也让我们认出了哪些情况下会发生溢出。当 x 和 y 都是负数，但是 $x+^t_w y\geq 0$ 时，我们就会得到负溢出。当 x 和 y 都是正数，但是 $x+^t_w y<0$ 时，我们会得到正溢出。

x	y	$x+y$	$x+^t_4 y$	情况
−8 [1000]	−5 [1011]	−13	3 [0011]	1
−8 [1000]	−8 [1000]	−16	0 [0000]	1
−8 [1000]	5 [0101]	−3	−3 [1101]	2
2 [0010]	5 [0101]	7	7 [0111]	3
5 [0101]	5 [0101]	10	−6 [1010]	4

图 2.18 二进制补码加法示例

可以通过执行运算数的二进制加法并将结果截断到四位，来获得四位二进制补码和的位级表示。

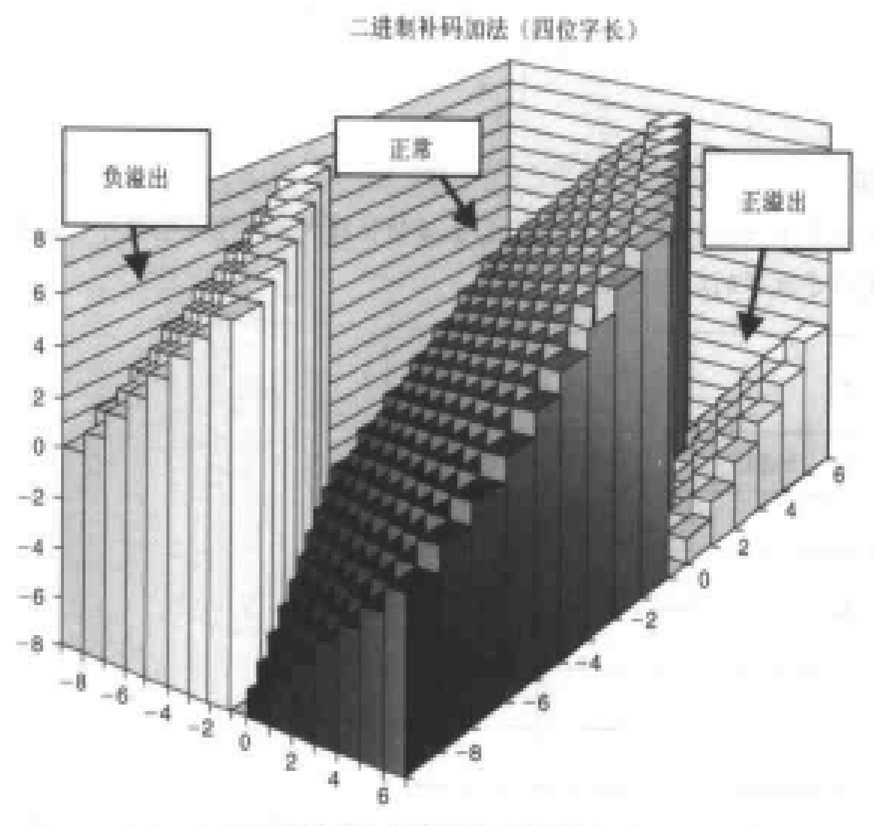


图 2.19 二进制补码加法

在字长为四位的情况下，当 $x+y<-8$ 时，加法会产生负溢出，而当 $x+y\geq 8$ 时，会产生正溢出。

练习题 2.25

按照图 2.18 的风格填写下表。请给出五位参数的整数值、它们的整数和二进制补码和的数值、二进制补码和的位级表示，以及根据等式 2.12 推导的情况。

x	y	$x+y$	$x+^t_4 y$	情况
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

2.3.3 二进制补码的非

我们可以看到范围$-2^{w-1} \le x < 2^{w-1}$中的每个数字 x 都有$+^t_w$下的加法逆元：首先，对于$x \ne -2^{w-1}$，我们可以看到它的加法逆元就是$-x$。也就是，我们有$-2^{w-1} < -x < 2^{w-1}$和$-x +^t_w x = -x + x = 0$。另一方面，对于$x = -2^{w-1} = TMin_w$，$-x = 2^{w-1}$不能被表示为一个 w 位的数。我们声明，这个特殊值本身就是它在$+^t_w$下的加法逆元。$-2^{w+1} +^t_w -2^{w+1}$的值由等式2.12的第三种情况给出，因为$-2^{w-1} + -2^{w-1} = -2^w$。这得到$-2^{w+1} +^t_w -2^{w+1} = -2^w + 2^w = 0$。从这个分析中，我们可以定义对于范围$-2^{w-1} \le x < 2^{w-1}$内的 x，二进制补码的非运算（negation operation）如下：

$$-^t_w x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \tag{2.13}$$

练习题 2.26

我们可以用一个十六进制数字来表示长度 w=4 的位模式。对于这些数字的二进制补码的解释，填写下表，确定所示数字的加法逆元。

x		$-^t_4 x$	
十六进制	十进制	十进制	十六进制
0			
3			
8			
A			
F			

对于二进制补码和无符号（练习题 2.24）非（negation）产生的位模式，你观察到什么？

一种有名的用来执行位级二进制补码的非（negation）的技术是，对每个位取反（或取补），然后将结果加 1。在 C 中，这可以写成~x + 1。为了验证这种技术的正确性，可以观察，对于每个位 x_i，我们有$\sim x_i = 1 - x_i$。设$\vec{x}$是一个长度为 w 的位向量，$x \doteq B2T_w(\vec{x})$是它表示的二进制补码数。根据等式 2.2，取反了的位向量$\sim\vec{x}$有如下数值：

$$
\begin{aligned}
B2T_w(\sim\vec{x}) &= -(1-x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1-x_i)2^i \\
&= \left[-2^{w-1} + \sum_{i=0}^{w-2}2^i\right] - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2}x_i 2^i\right] \\
&= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\
&= -1 - x
\end{aligned}
$$

上述推导中，关键的简化是 $\sum_{i=0}^{w-2}2^i = 2^{w-1}-1$。只要将 $\sim\vec{x}$ 加 1，我们就能得到 $-x$ 了。

要将位级表示为 $\vec{x} \doteq [x_{w-1}, x_{w-2}, \cdots, x_0]$ 的数 x 加 1，将运算 incr 定义为：设 k 为最右边的 0 的位置，这样 $\vec{x}$ 就具有这样的形式 $[x_{w-1}, x_{w-2}, \cdots, x_{k+1}, 0, 1, \cdots, 1]$，然后，我们将 incr($\vec{x}$)定义为 $[x_{w-1}, x_{w-2}, \cdots, x_{k+1}, 1, 0, \cdots, 0]$。对于 x 的位级表示为 $[1, 1, \cdots, 1]$ 的特殊情况，将 incr($\vec{x}$)定义为 $[0, \cdots, 0]$。为了说明 incr($\vec{x}$) 得到的是 $x +^t_w 1$ 的位级表示，考虑下面的情况：

1. 当 $\vec{x} = [1, 1, \cdots, 1]$ 时，我们有 $x = -1$。被增加的值 incr($\vec{x}$) $\doteq [0, \cdots, 0]$ 有数值 0。

2. 当 $k = w-1$ 时，即 $\vec{x} = [0, 1, \cdots, 1]$ 时，我们有 $x = TMax_w$。被增加的值 incr($\vec{x}$) $= [1, 0, \cdots, 0]$ 有数值 $TMin_w$。从等式 2.12，我们可以看到 $TMax_w +^t_w 1$ 是正溢出的情况之一，得到 $TMin_w$。

3. 当 $k < w-1$ 时，也就是 $x \neq TMax_w$ 且 $x \neq -1$ 时，我们可以看出 incr($\vec{x}$) 的低 $k+1$ 位有数值 2^k，而 $\vec{x}$ 的低 $k+1$ 位的数值为 $\sum_{i=0}^{k-1}2^i = 2^k - 1$。它们的高 $w-k+1$ 位有相同的数值。因此，incr($\vec{x}$) 有数值 $x+1$。此外，对于 $x \neq TMax_w$，对 x 加 1 不会导致溢出，因此 $x +^t_w 1$ 也等于 $x+1$。

正如说明的那样，图 2.20 展示了取反（或取补）和加 1 是如何影响几个四位向量的数值的。

$\vec{x}$		$\sim\vec{x}$		incr($\sim\vec{x}$)	
[0101]	5	[1010]	-6	[1011]	-5
[0111]	7	[1000]	-8	[1001]	-7
[1100]	-4	[0011]	3	[0100]	4
[0000]	0	[1111]	-1	[0000]	0
[1000]	-8	[0111]	7	[1000]	-8

图 2.20 取反（或取补）和增加四位数字的示例

效果就是计算二的值的非。

2.3.4 无符号乘法

范围 $0 \leq x, y \leq 2^w-1$ 内的整数 x 和 y 可以被表示为 w 位的无符号数字，但是它们的乘积 $x \cdot y$ 的取值范围为 $0 \sim (2^w-1)^2 = 2^{2w}-2^{w+1}+1$ 之间。这可能需要 $2w$ 位来表示。不过，C 中的无符号乘法被定义为产生 $2w$ 位的整数乘积的低 w 位表示的值。根据等式 2.7，我们可以看出这等价于计算模 2^w 的乘法。因此，w 位无符号乘法运算 $*^u_w$ 的效果为

$$x *^u_w y = (x \cdot y) \bmod 2^w \tag{2.14}$$

大家都知道模运算形成了环。因此我们可以推出 w 位数字上的无符号运算形成了环 $\langle\{0, \cdots, 2^w-1\},$

$+^u_w, *^u_w, -^u_w, 0, 1\rangle$。

2.3.5 二进制补码乘法

范围$-2^{w-1} \le x, y \le 2^{w-1}-1$内的整数 x 和 y 可以被表示为 w 位的二进制补码数字，但是它们的乘积 $x \cdot y$ 的取值范围为$-2^{w-1} \cdot (-2^{w-1}-1) = -2^{2w-2}+2^{w-1} \sim -2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$之间。要想用二进制补码来表示这个乘积，可能需要 $2w$ 位——大多数情况下只需要 $2w-1$ 位，但是特殊情况 2^{2w-2} 需要 $2w$ 位（包括一个符号位 0）。然而，C 中的有符号乘法是通过将 $2w$ 位的乘积截断为 w 位来实现的。根据等式 2.8，w 位的二进制补码乘法运算 $*^t_w$ 的效果为：

$$x *^t_w y = U2T_w((x \cdot y) \bmod 2^w) \tag{2.15}$$

我们认为对于无符号和二进制补码乘法来说，乘法运算的位级表示都是一样的。也就是，给定长度为 w 的位向量 $\vec{x}$ 和 $\vec{y}$，无符号乘积 $B2U_w(\vec{x}) *^u_w B2U_w(\vec{x})$ 的位级表示与二进制补码乘积 $B2T_w(\vec{x}) *^t_w B2T_w(\vec{x})$ 的位级表示相同。这表明机器可以用一种乘法指令来进行有符号和无符号整数的乘法。

为了看清这一点，设 $x = B2T_w(\vec{x})$ 和 $y = B2T_w(\vec{y})$ 是这些位模式表示的二进制补码值，而 $x' = B2U_w(\vec{x})$ 和 $y' = B2U_w(\vec{y})$ 是这些位模式表示的无符号值。根据等式 2.3，我们有 $x' = x + x_{w-1}2^w$ 和 $y' = y + y_{w-1}2^w$。计算这些值的模 2^w 乘积得到以下结果：

$$\begin{aligned}(x' \cdot y') \bmod 2^w &= [(x+x_{w-1}2^w) \cdot (y+y_{w-1}2^w)] \bmod 2^w \\ &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\ &= (x \cdot y) \bmod 2^w \end{aligned} \tag{2.16}$$

因此，$x \cdot y$ 和 $x' \cdot y'$ 的低 w 位是相同的。

正如说明的那样，图 2.21 展示了不同的三位数字乘法的结果。对于每对位级运算数，我们既执行无符号乘法，也执行有符号乘法。注意，无符号已截断乘积总是等于 $x \cdot y \bmod 8$，而且两个已截断乘积的位级表示是相同的。

模式	x		y		$x \cdot y$		截断的 $x \cdot y$	
无符号	5	[101]	3	[011]	15	[001111]	7	[111]
二进制补码	-3	[101]	3	[011]	-9	[110111]	-1	[111]
无符号	4	[100]	7	[111]	28	[011100]	4	[100]
二进制补码	-4	[100]	-1	[111]	4	[000100]	-4	[100]
无符号	3	[011]	3	[011]	9	[001001]	1	[001]
二进制补码	3	[011]	3	[011]	9	[001001]	1	[001]

图 2.21 三位无符号和二进制补码乘法示例

虽然完整的乘积的位级表示可能会不同，但是已截断乘积的位级表示是相同的。

练习题 2.27

填写下表，说明不同的三位数字乘法的结果，按照图 2.21 的风格：

模式	x	y	$x \cdot y$	截断的 $x \cdot y$
无符号 二进制补码	[110] [110]	[010] [010]		
无符号 二进制补码	[001] [001]	[111] [111]		
无符号 二进制补码	[111] [111]	[111] [111]		

我们可以看出，w 位数字上的无符号运算和二进制补码运算是同构的——运算 $+^u_w$、$-^u_w$、$*^u_w$ 和 $+^t_w$、$-^t_w$、$*^t_w$ 有相同的位级效果。据此，我们可以推断出二进制补码运算形成了环 $\langle\{-2^{w-1},\cdots,2^{w-1}-1\}, +^t_w, *^t_w, -^t_w, 0, 1\rangle$。

2.3.6 乘以 2 的幂

在大多数机器上，整数乘法指令相当地慢，需要 12 或者更多的时钟周期，然而其他整数运算——例如加法、减法、位级运算和移位——只需要 1 个时钟周期。因此，编译器使用的一项重要的优化就是试着用移位和加法运算的组合来代替乘以常数因子的乘法。

设 x 为位模式$[x_{w-1}, x_{w-2},\cdots, x_0]$表示的无符号整数。那么，对于任何 $k \geq 0$，我们都认为 $x2^k$ 的位级表示是由$[x_{w-1}, x_{w-2},\cdots, x_0, 0,\cdots, 0]$给出的，这里右边增加了 k 个 0。这个属性可以通过等式 2.1 推导出来：

$$
\begin{aligned}
B2U_{w+k}([x_{w-1}, x_{w-2},\cdots,x_0,0,\cdots,0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\
&= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \cdot 2^k \\
&= x2^k
\end{aligned}
$$

对于 $k \geq w$，我们可以将移位了的位向量截断到长度 w，得到$[x_{w-k-1}, x_{w-k-2},\cdots, x_0, 0,\cdots, 0]$。根据等式 2.7，这个位向量的数值为 $x2^k \bmod 2^w = x *^u_w 2^k$。因此，对于无符号变量 x，C 表达式 x<<k 等价于 x * pwr2k，这里 pwr2k 等于 2^k。特别地，我们可以用 1U<<k 来计算 pwr2k。

通过类似的推理，我们可以给出，对于一个位模式为$[x_{w-1}, x_{w-2},\cdots, x_0]$的二进制补码数 x，以及范围 $0 \leq k < w$ 内任意的 k，位模式$[x_{w-k-1}, x_{w-k-2},\cdots, x_0, 0,\cdots, 0]$就是 $x *^t_w 2^k$ 的二进制补码表示。因此，对于有符号变量 x，C 表达式 x<<k 等价于 x*pwr2k，这里 pwr2k 等于 2^k。

注意，无论是无符号运算还是二进制补码运算，乘以 2 的幂都可能会导致溢出。我们的结果表明，即使溢出的时候，我们通过移位得到的结果也是一样的。

练习题 2.28

就像我们将在第 3 章中看到的那样，Intel 兼容的处理器上的 leal 指令能够执行 a<<k+b 形式的计算，这里 k 或者等于 0、1 或 2，而 b 等于 0，或者等于某个程序值。编译器常常用这条指令来执行常数因子乘法。例如，我们可以用 a<<1+a 来计算 3*a。

用这条指令可以计算 a 的哪些倍数?

2.3.7 除以 2 的幂

在大多数机器上，整数除法要比整数乘法更慢——需要 30 或者更多的时钟周期。除以 2 的幂也可以用移位运算来实现，只不过我们用的是右移，而不是左移。对于无符号和二进制补码数，分别使用逻辑移位和算术移位来达到目的。

整数除法总是舍入到 0 的。对于 $x \geq 0$ 和 $y > 0$，结果会是$\lfloor x/y \rfloor$，这里对于任何实数 a，$\lfloor a \rfloor$定义为惟一的整数a'，使得$a' \leq a < a' + 1$。例如，$\lfloor 3.14 \rfloor = 3$，$\lfloor -3.14 \rfloor = -4$ 和$\lfloor 3 \rfloor = 3$。

考虑在一个无符号数上执行逻辑右移的效果。设 x 为位模式$[x_{w-1}, x_{w-2}, \cdots, x_0]$表示的无符号整数，而 k 的取值范围为 $0 \leq k < w$。设x'为 $w-k$ 位表示$[x_{w-1}, x_{w-2}, \cdots, x_k]$的无符号数，而$x''$为 k 位表示$[x_{k-1}, \cdots, x_0]$的无符号数。我们有$x' = \lfloor x/2^k \rfloor$。证明如下：根据等式 2.1，我们有 $x = \sum_{i=0}^{w-1} x_i 2^i$，$x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ 和 $x'' = \sum_{i=0}^{k-1} x_i 2^i$。因此，我们可以把 x 写为 $x = 2^k x' + x''$。可以观察到$0 \leq x'' \leq = \sum_{i=0}^{k-1} 2^i = 2^k - 1$，因此$0 \leq x'' < 2^k$，这意味着$\lfloor x''/2^k \rfloor = 0$。因此，$\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' \lfloor x''/2^k \rfloor = x'$。

可以观察到，对位向量$[x_{w-1}, x_{w-2}, \cdots, x_0]$逻辑右移 k 位会得到位向量

$$[0, \cdots, 0, x_{w-1}, x_{w-2}, \cdots, x_k]$$

这个位向量有数值x'。也就是，将一个无符号数逻辑右移 k 位等价于把它除以 2^k。因此，对于无符号变量 x，C 表达式 x>>k 等价于 x/pwr2k，这里 pwr2k 等价于 2^k。

现在考虑对一个二进制补码数进行算术右移的结果。设 x 为位模式$[x_{w-1}, x_{w-2}, \cdots, x_0]$表示的二进制补码整数，而 k 的取值范围为 $0 \leq k < w$。设 x' 为 $w-k$ 位$[x_{w-1}, x_{w-2}, \cdots, x_k]$表示的二进制补码数，而 x'' 为低 k 位$[x_{k-1}, \cdots, x_0]$表示的无符号数。通过对无符号情况的类似分析，我们有 $x = 2^k x' + x''$，而 $0 \leq x'' < 2^k$，得到$x' = \lfloor x/2^k \rfloor$。此外，我们可以观察到，算术右移位向量$[x_{w-1}, x_{w-2}, \cdots, x_0]$ k 位，得到位向量

$$[x_{w-1}, \cdots, x_{w-1}, x_{w-1}, x_{w-2}, \cdots, x_k]$$

它刚好就是将$[x_{w-1}, x_{w-2}, \cdots, x_k]$从 $w-k$ 位符号扩展到 w 位。因此，这个移位了的位向量就是$\lfloor x/y \rfloor$的二进制补码表示。

对于 $x \geq 0$，我们的分析表明这个移位的结果就是所期望的值。不过，对于 $x < 0$ 和 $y > 0$，整数除法的结果应该是$\lceil x/y \rceil$，这里，对于任何实数 a，$\lceil a \rceil$被定义为使得$a' - 1 < a \leq a'$的惟一整数a'。也就是说，整数除法应该将为负的结果向上朝零舍入。例如，C 表达式-5/2 得到-2。因此，当有舍入发生时，将一个负数右移 k 位不等价于把它除以 2^k。例如，-5 的四位表示为[1011]。如果我们将它算术右移一位，我们得到[1101]，这是-3 的二进制补码表示。

我们可以通过在移位之前“偏置（biasing）”这个值，修正这种不合适的舍入。这种技术利用的是这样一个属性：对于整数 x 和有 $y > 0$ 的 y，$\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$。因此，对于 $x < 0$，如果我们在右移之前，先将 x 加上 2^k-1，那么我们就会得到正确舍入的结果了。这个分析表明对于使用算术右移的二进制补码机器，C 表达式

```
(x<0 ? (x - (1<<k) - 1) : x) >> k
```

等价于 x/pwr2k，这里 pwr2k 等于 2^k。例如，-5 除以 2，我们先加上偏置数 2-1 = 1，得到位模式[1100]。

将这个值算术右移1位得到位模式[1110]，这是-2的二进制补码表示。

练习题 2.29

在下面的代码中，我们省略了常数M和N的定义：

```
#define M /* Mystery number 1 */
#define N /* Mystery number 2 */
int arith(int x, int y)
{
  int result = 0;
  result = x*M + y/N; /* M and N are mystery numbers. */
  return result;
}
```

我们以某个M和N的值编译这段代码。编译器用我们讨论过的方法优化乘法和除法。下面是将产生出的机器代码翻译回C语言的结果：

```
/* Translation of assembly code for arith */
int optarith(int x, int y)
{
  int t = x;
  x <<= 4;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2; /* Arithmetic shift */
  return x+y;
}
```

M和N的值为多少？

练习题 2.30

假设我们在对有符号值使用二进制补码运算的32位机器上运行代码。对于有符号值使用的是算术右移，而对于无符号值使用的是逻辑右移。变量的声明和初始化如下：

```
int x = foo();    /* Arbitrary value */
int y = bar();    /* Arbitrary value */

unsigned ux = x;
unsigned uy = y;
```

对于下面每个C表达式，或者证明对于所有的x和y值，它都为真（等于1），或者给出使得它为假（等于0）的x和y的值：

```
A. (x >= 0) || ((2*x) < 0)
B. (x & 7) != 7 || (x<<30 < 0)
C. (x * x) >= 0
D. x < 0 || -x <= 0
E. x > 0 || -x >= 0
```

```
F. x*y == ux*uy
G. .x*y - uy*ux == -y
```

2.4 浮点

浮点表示对形如 $V=x\times 2^y$ 的有理数进行编码。它对执行含有非常大的数字（$|V|>>0$）、非常接近于0（$|V|<<1$）的数字，以及更普遍地作为实数运算的近似值的计算，是很有用的。

直到20世纪80年代，每个计算机制造商都设计了自己的表示浮点数的规则，以及对浮点数执行运算的细节。另外，他们常常不会太多地关注运算的精确性，而把实现的速度和简便性看得比数字精确性更重要。

大约在1985年，这些情况随着IEEE标准754的推出而改变了，这是一个仔细制订的表示浮点数及其运算的标准。这项工作是从1976年Intel发起8087的设计开始的，8087是一种为8086处理器提供浮点支持的芯片。他们雇佣了William Kahan，加州大学伯克利分校的一位教授，作为帮助设计未来处理器浮点标准的顾问。他们支持Kahan加入一个IEEE资助的制订工业标准的委员会。这个委员会最终采纳了一个非常接近于Kahan为Intel设计的标准。目前，实际上所有的计算机都支持这个后来被称为IEEE浮点的标准。这大大改善了科学应用程序在不同机器上的可移植性。

旁注：IEEE（电气和电子工程师协会）

电气和电子工程师协会（IEEE——读做"I-Triple-E"）是一个包括所有电子和计算机技术的专业团体。它出版刊物、举办会议，并且建立协会团体来定义标准，内容涉及从电力传输到软件工程。

在本节中，我们将看到IEEE浮点格式中数字是如何被表示的。我们还将探讨舍入（rounding）的问题，当一个数字不能被准确地表示为这种格式，因此必须被向上调整或者向下调整时，就会出现舍入。然后，我们将探讨加法、乘法和关系运算符的数学属性。许多程序员认为浮点最没意思，而且最深奥难懂。我们将看到，因为IEEE格式是定义在一组小而一致的原则上的，所以它实际上是相当优雅和容易理解的。

2.4.1 二进制小数

理解浮点数的第一步是考虑含有小数值的二进制数字。首先，让我们来看看更熟悉的十进制表示法。十进制表示法使用这样形式的表示：$d_m d_{m-1}\cdots d_1 d_0 . d_{-1} d_{-2}\cdots d_{-n}$，其中每个十进制数 d_i 的取值范围在0～9之间。这个表达式描述的数 d 定义如下：

$$d = \sum_{i=-n}^{m} 10^i \times d_i$$

数字的权被定义为和十进制小数点符号"."相关，这意味着点左边的数字的权是10的正幂，得到整数值，而点右边的数字的权是10的负幂，得到小数值。例如，12.34_{10} 表示数字 $1\times 10^1+2\times 10^0+3\times 10^{-1}+4\times 10^{-2}=12\frac{34}{100}$。

类似地，考虑一个形如 $b_m b_{m-1}\cdots b_1 b_0 . b_{-1} b_{-2}\cdots b_{-n}$ 的表示法，其中每个二进制数字，或者称为位，b_i 的取值范围是在0～1之间。这种表示方法表示的数 b 定义如下：

$$b = \sum_{i=-n}^{m} 2^i \times b_i \tag{2.17}$$

符号“.”现在变为了二进制的点，点左边的位的权是 2 的正幂，点右边的位是 2 的负幂。例如，101.11_2 表示数字 $1\times2^2+0\times2^1+1\times2^0+1\times2^{-1}+1\times2^{-2}=4+0+1+\frac{1}{2}+\frac{1}{4}=5\frac{3}{4}$。

从等式 2.17 中可以很容易地看出，向左移动二进制小数点一位相当于这个数被 2 除。例如，101.11_2 表示数 $5\frac{3}{4}$，而 10.111_2 表示数 $2+0+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}=2\frac{7}{8}$。类似地，向右移动二进制小数点一位相当于将该数乘 2。例如 1011.1_2 表示数 $8+0+2+1+\frac{1}{2}=11\frac{1}{2}$。

注意形如 $0.11\cdots1_2$ 的数表示的是刚好小于 1 的数。例如，0.111111_2 表示 $\frac{63}{64}$，我们将用简单的表达法 $1.0-\varepsilon$ 来表示这样的数值。

假定我们仅考虑有限长度的编码，那么十进制符号是不能准确地表达像 $\frac{1}{3}$ 和 $\frac{5}{7}$ 这样的数的。类似地，小数的二进制表示法只能表示那些能够被写成 $x \times 2^y$ 的数。其他的值只能够被近似地表示。例如，虽然加长二进制表示能够提高近似表示数 $\frac{1}{5}$ 的精度，但是我们并不能把它准确地表示为一个二进制数：

表示	值	十进制
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

练习题 2.31

填写下表中的缺失的信息：

小数值	二进制表示	十进制表示
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.375
		3.0625

练习题 2.32

浮点运算的不精确性能够产生灾难性的后果。1991年2月25日，在海湾战争期间，沙特阿拉伯的达摩地区设置的美国爱国者导弹，拦截伊拉克的飞毛腿导弹失败。飞毛腿导弹击中了美国的一个兵营，造成28名士兵死亡。美国总审计局（GAO）对失败原因做了详细的分析[52]，并且确定潜在的原因在于一个数字计算不精确。在这个练习中，你将重现总审计局分析的一部分。

爱国者导弹系统中含有一个内置的时钟，实现为一个计数器，每0.1秒就加1。为了以秒为单位来确定时间，程序将用一个24位的近似于$\frac{1}{10}$的二进制小数值来乘以这个计数器的值。特别地，$\frac{1}{10}$的二进制表达式是无穷序列

$$0.000110011[0011]\cdots_2$$

其中，方括号里的部分是无限重复的。计算机只用这个序列的开头位和二进制小数点右边的头23位来近似地表示0.1。我们称这个数为x。

A. x-0.1 的二进制表示是什么？

B. x-0.1 的近似的十进制值是多少？

C. 当系统初始启动时，时钟从0开始，并且一直保持计数，在这个例子中，系统已经运行了大约100个小时。程序计算的时间和实际的时间之差为多少？

D. 系统根据一枚来袭的的导弹的速率和它最后被雷达侦测到的时间，来预测它将在哪里出现。假定飞毛腿的速率大约是2000米每秒，对它的预测偏差了多少？

正常地，一个通过一次读取时钟得到的绝对时间中的轻微错误不会影响跟踪的计算。反而，它应该依赖于两次连续的读取的之间的相对时间。问题是爱国者导弹的软件已经升级成使用更精确的函数来读取时间，但是不是所有的函数调用都用新的代码替换了。结果就是，跟踪软件使用了一次读取的是精确的时间，但其他软件读取的是不精确的时间[71]。

2.4.2 IEEE 浮点表示

像前一节中谈到的位置表示法不能很有效地表示非常大的数字。例如，表达式 5×2^{100} 的表示是由 101 后面跟随 100 个零的位模式组成的。相反地，我们希望通过给定 x 和 y 的值，来表示形如 $x\times2^y$ 的数。

IEEE 浮点标准用 $V=(-1)^s\times M\times 2^E$ 的形式来表示一个数：

- 符号（sign）s 决定数是负数（s=1）还是正数（s=0），而对于数值 0 的符号位解释作为特殊情况处理。
- 有效数（significand）M 是一个二进制小数，它的范围在 1～2-ε 之间，或者在 0～1-ε 之间。
- 指数（exponent）E 是 2 的幂（可能是负数），它的作用是对浮点数加权。

浮点数的位表示被划分为三个域，以编码这些值：

- 一个单独的符号位 s 直接编码符号 s。
- k 位的指数域 $\exp=e_{k-1}\cdots e_1e_0$ 编码指数 E。
- n 位小数域 frac $=f_{n-1}\cdots f_1f_0$ 编码有效数 M，但是被编码的值也依赖于指数域的值是否等于零。

在单精度浮点格式（C 语言中的 float）中，s、exp 和 frac 域分别为 1 位、k=8 位和 n=23 位，产生一个 32 位的表示。在双精度浮点格式（C 语言中的 double）中，s、exp 和 frac 域分别为 1 位、k=11 位和 n=52 位，产生一个 64 位的表示。

给定位表示，根据 exp 的值，被编码的值可以分成三种不同的情况。

规格化值

这是最普遍的情况。当 exp 的位模式既不是全为 0（数值 0），也不是全为 1（单精度数值为 255，双精度数值为 2047）时，就都属于这类情况。在这种情况中，指数域解释为表示偏置（biased）形式的有符号整数。也就是说，指数的值是 $E=e-Bias$，其中 e 是无符号数，其位表示为 $e_{k-1}\cdots e_1e_0$，而 $Bias$ 是一个等于 $2^{k-1}-1$（单精度是 127，双精度是 1023）的偏置值。由此产生了指数的取值范围，对于单精度是-126～+127，而对于双精度是-1022～+1023。

小数域 frac 解释为描述小数值 f，其中 $0\le f<1$，其二进制表示为 $0.f_{n-1}\cdots f_1f_0$，也就是二进制小数点在最高有效位的左边。有效数定义为 $M=1+f$。有时，这种方式也叫做*隐含的以 1 为开头的*（implied leading 1）表示，因为我们可以把 M 看成一个二进制表达式为 $1.f_{n-1}f_{n-2}\cdots f_0$ 的数字。既然我们总是能够调整指数 E，使得有效数 M 在范围 $1\le M<2$ 之中（假设没有溢出），那么这种表示方法是一种免费获得一个额外精度位的技巧。既然第一位总是等于 1，那么我们就不需要显式地来表示它了。

非规格化值

当指数域为全 0 时，所表示的数就是非规格化形式的。在这种情况下，指数值是 $E=1-Bias$，而有效数的值是 $M=f$，也就是小数域的值，不包含隐含的开头的 1。

旁注：为什么对于非规格化值要这样设置偏置值？

使指数值为 1-Bias 而不是简单的-Bias 似乎是违反直觉的。我们将很快看到，这种方式提供了一种从非规格化值平滑转换到规格化值的方法。

非规格化数有两个目的。首先，它们提供了一种表示数值 0 的方法，因为使用规格化数，我们

必须总是使 $M \geq 1$，因此我们就不能表示0。实际上，+0.0的浮点表示的位模式为全0：符号位是0，指数域全为0（表明是一个非规格化值），而小数域也全为0，这就得到 $M=f=0$。令人奇怪的是，当符号位为1，但是其他域全为0时，我们得到值-0.0。根据IEEE的浮点格式，值+0.0和-0.0在某些方面被认为是不同的，而在其他方面是相同的。

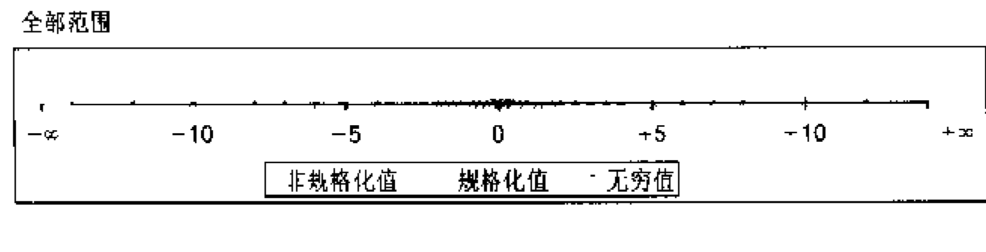
非规格化数的另外一个功能是用来表示那些非常接近于0.0的数。它们提供了一种属性，称为逐渐溢出（gradual underflow），其中，可能的数值分布均匀地接近于0.0。

特殊数值

最后一类数值是当指数域全为1的时候出现的。当小数域全为0时，得到的值表示无穷，当 $s=0$ 时是 $+\infty$，或者当 $s=1$ 时是 $-\infty$。当我们把两个非常大的数相乘，或者我们除零时，无穷能够表示溢出的结果。当小数域为非零时，结果值被称为"NaN"，就是"不是一个数（Not a Number）"的缩写。一些运算的结果不能是实数或无穷，就会返回这样的NaN值，比如当计算 $\sqrt{-1}$ 或 $\infty-\infty$ 时。在某些应用中，用来表示未初始化的数据，它们也很有用处。

2.4.3 数值示例

图2.22展示了一组数值，它们可以用假定的6位格式来表示，有 $k=3$ 的指数位和 $n=2$ 的有效数位。偏置量是 $2^{3-1}-1=3$。图中的A部分显示了所有可表示的值（除了NaN）。两个无穷值在两个末端。规格化数具有的最大数量级是±14。非规格化数聚集在0的附近，在图的B部分中，我们只展示了介于-1.0和+1.0之间的数值，这样这部分就能够看得更加清楚了。两个零是特殊的非规格化数。可以观察到，那些可表示的数并不是均匀分布的——它们在越靠近原点处越稠密。



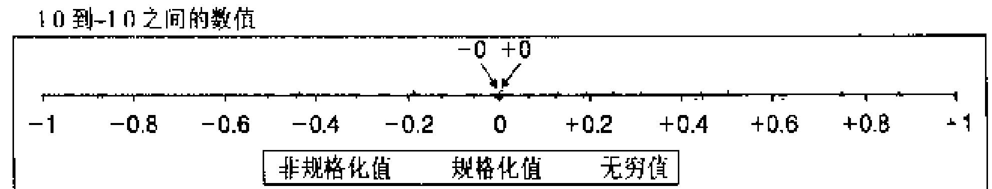


图2.22 6位浮点格式可表示的值

有 $k=3$ 的指数位和 $n=2$ 的有效数位。偏置量是3。

图2.23展示了假定的8位浮点格式的示例，其中有 $k=4$ 的指数位和 $n=3$ 的小数位。偏置量是 $2^{4-1}-1=7$。图被分成了三个区域，来描述三类数字。从0开始，最靠近0的是非规格化数。这种格式的非规格化数的 $E=1-7=-6$，得到 $2^E=\frac{1}{64}$。小数f的值的范围是 $0,\frac{1}{8},\cdots,\frac{7}{8}$，从而得到数 V 的范围是 $0\sim\frac{7}{8\times 64}=\frac{7}{512}$。

描　述	位表示	e	E	f	M	V
0	0 0000 000	0	6	0	0	0
最小的非规格化数	0 0000 001	0	-6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	0 0000 010	0	-6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	0 0000 011	0	-6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
				…		
	0 0000 110	0	-6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
最大的非规格化数	0 0000 111	0	-6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
最小的规格化数	0 0001 000	1	-6	0	$\frac{8}{8}$	$\frac{8}{512}$
	0 0001 001	1	-6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
				…		
	0 0110 110	6	-1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	0 0110 111	6	-1	$\frac{7}{8}$	$\frac{14}{8}$	$\frac{15}{16}$
1	0 0111 000	7	0	0	$\frac{8}{8}$	1
	0 0111 001	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	0 0111 010	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$
				…		
	0 1110 110	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
最大的规格化数	0 1110 111	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
无穷大	0 1111 000	—	—	—	—	$+\infty$

图 2.23　8 位浮点格式的非负值示例

有 $k=4$ 的指数位的和 $n=3$ 的有效数位。偏置量是 7。

这种形式的最小规格化数同样有 $E=1-7=-6$，并且小数取值范围也为 $0,\frac{1}{8},\cdots,\frac{7}{8}$。然而，有效数在范围 $1+0=1\sim1+\frac{7}{8}=\frac{15}{8}$ 之间，得出数 V 在范围 $\frac{8}{512}\sim\frac{12}{512}$ 之间。

可以观察到最大非规格化数 $\frac{7}{512}$ 和最小规格化数 $\frac{8}{512}$ 之间的平滑转变。这种平滑性归功于我们对非规格化数 E 的定义。通过将 E 定义为 1-*Bias*，而不是 -*Bias*，这样我们可以补偿非规格化数的有效数没有隐含的开头的 1。

当我们增大指数时，我们成功地得到更大的规格化值，经过 1.0，然后得到最大的规格化数。这个数具有指数 $E=7$，得到一个权 $2^E=128$。小数等于 $\frac{7}{8}$ 得到有效数 $M=\frac{15}{8}$。因此，数值是 $V=240$。超出这个值就会溢出到 $+\infty$。

这个表达式具有一个有趣属性，假如我们将图 2.23 中的值的位表达式解释为无符号整数，它们就是按升序排列的，就像它们表示的浮点数一样。这不是偶然的——IEEE 格式如此设计就是为了浮点数能够使用整数排序函数来进行排序。当处理负数时，有一个小的难点，因为它们有开头的 1，并且它们是按照降序出现的，但是不需要浮点运算来进行比较也能解决这个问题（参见练习题 2.56）。

练习题 2.33

假设一个基于 IEEE 浮点格式的 5 位浮点表示，有 1 个符号位、2 个指数位（$k=2$）和两个小数位（$n=2$）。指数偏置量是 $2^{2-1}-1=1$。

下表中列举了这个 5 位浮点表示的全部非负取值范围。使用下面的条件，填写表格中的空白项：

e：假定指数域是一个无符号整数所表示的值。

E：偏置之后的指数值。

f：小数值。

M：有效数的值。

V：被表示的数字值。

用形如 $\frac{x}{4}$ 的小数表示 f、M 和 V 的值。被"——"标注的条目不用填。

位	*e*	*E*	*f*	*M*	*V*
0 00 00					
0 00 01					
0 00 10					
0 00 11					
0 01 00					
0 01 01					
0 01 10					

（续表）

位	e	E	f	M	V
0 01 11					
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01					
0 10 10					
0 10 11					
0 11 00	—	--	—	—	+∞
0 11 01	—	—	—	—	*NaN*
0 11 10	·	—	—	—	*NaN*
0 11 11	—	—	—	—	*NaN*

图 2.24 展示了一些重要的单精度和双精度浮点数的表示和数字值根据图 2.23 中展示的 8 位格式，我们能够看出有 k 位指数和 n 位小数的浮点表示的一般属性：

描 述	指数	小数	单精度		双精度	
			值	十进制	值	十进制
0	00…00	0…00	0	0.0	0	0.0
最小非规格化数	00…00	0…01	$2^{-23}\times2^{-126}$	1.4×10^{-45}	$2^{-52}\times2^{1022}$	4.9×10^{324}
最大非规格化数	00…00	1…11	$(1-\varepsilon\times2^{125}$	1.2×10^{-38}	$(1-\varepsilon)\times2^{-1022}$	2.2×10^{-308}
最小规格化数	00…01	0…00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01…11	0…00	1×2^{0}	1.0	1×2^{0}	1.0
最大规格化数	11…10	1…11	$(2-\varepsilon)\times2^{127}$	3.4×10^{38}	$(2-\varepsilon)\times2^{1023}$	1.8×10^{308}

图 2.24 非负浮点数的示例

- 值+0.0 总有一个全为 0 的位表示。
- 最小的正（positive）非规格化值有一个位表示，是由最低有效位为 1 而其他所有位为 0 构成的。它具有小数（和有效数）值 $M=f=2^{-n}$ 和一个指数值 $E=-2^{k-1}+2$。因此它的数字值是 $V=2^{-n-2^{k-1}+2}$。
- 最大的非规格化值的位模式是由全为 0 的指数域和全为 1 的小数域组成的。它有小数（和有效数）值 $M=f=1-2^{-n}$（我们写成 1-ε）和指数值 $E=-2^{k-1}+2$。因此，数值 $V=(1-2^{-n})\times2^{-2^{k-1}+2}$，这仅比最小的规格化值小一点。
- 最小的正（positive）规格化值的位模式的指数域的最低有效位为 1，其他位全为 0。它的有效数值 $M=1$，而指数值 $E=-2^{k-1}+2$。因此，数值 $V=2^{-2^{k-1}+2}$。
- 值 1.0 的位表示的指数域除了最高有效位等于 1 以外，其他位都等于 0。它的有效数值是 M=1，而它的指数值是 E=0。
- 最大的规格化值的位表示的符号位为 0，指数的最低有效位等于 0，其他位等于 1。它的小数值 $f=1-2^{-n}$，有效数 $M=2-2^{-n}$（我们写作 2-ε）。指数值 $E=2^{k-1}-1$，得到数值 $V=(2-2^{-n})$

$\times 2^{2^{k-1}-1} = (1-2^{-n-1}) \times 2^{2^{k-1}}$。

对理解浮点表示很有用的一个练习是把样本整数值转换成浮点形式。例如，在图 2.10 中我们看到 12 345 具有二进制表示[11000000111001]。通过向二进制小数点右边移动 13 位，我们创建这个数的一个规格化表示，得到 $12345 = 1.1000000111001_2 \times 2^{13}$。为了用 IEEE 单精度形式来编码，我们丢弃开头的 1，并且在末尾增加 10 个 0，来构造小数域，得到二进制表示 [10000001110010000000000]。为了构造指数域，我们增加偏置量 127 到 13，得到 140，其二进制表示为[10001100]。加上符号位 0，我们就得到二进制的浮点表示[01000110010000001110010000000000]。回想一下 2.1.4 节，我们观察到整数值 12345（0x3039）和单精度浮点值 12345.0（0x4640E400）在位级表示上有下列关系：

```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
                   *************
             4   6   4   0   E   4   0   0
          01000110010000001110010000000000
```

现在我们可以看到，相关的区域对应于整数的低位，刚好在最高的等于 1 的位之前停止（这个位就是隐含的开头的位 1），和浮点表示的小数部分的高位是相匹配的。

练习题 2.34

正如在练习题 2.6 中提到的，整数 3490593 的十六进制表示为 0x354321，而单精度浮点数 3490593.0 的十六进制表示为 0x4A550C84。推导出这个浮点表示，并解释整数和浮点数表示的位之间的关系。

练习题 2.35

A. 假定一个 k 位指数和 n 位小数的浮点格式，给出不能准确描述的最小正整数的公式（因为要想准确表示它需要 $n+1$ 位小数）。

B. 对于单精度格式（$k=8$，$n=23$），这个整数的数字值是多少？

2.4.4 舍入

因为表示方法限制了浮点数的范围和精度，所以浮点运算只能近似地表示实数运算。因此，对于值 x，我们一般想有一种系统的方法，能够找到“最接近的”匹配值 x'，它可以用期望的浮点形式表示出来。这就是舍入（rounding）运算的任务。关键问题是定义在两个可能值中间的数值的舍入方向。例如，如果我有 1.50 美元，想把它舍入到最接近的美元数，结果应该是选择 1 美元还是 2 美元呢？一种可选择的方法是维持实际数字的下界和上界。例如，我们可以确定可表示的值 x^- 和 x^+，使得 x 的值位于它们之间：$x^- \le x \le x^+$。IEEE 浮点格式定义了四种不同的舍入方式。默认的方法是找到最接近的匹配，而其他三种可用于计算上界和下界。

图 2.25 举例说明了四种舍入方式，将一个金额数舍入到最接近的整数上。向偶数舍入（round-to-even），也被称为向最接近的值舍入（round-to-nearest），是默认的方式。它试图找到一个最接近的匹配值。因此，它将 1.40 美元舍入成 1 美元，而将 1.60 美元舍入成 2 美元，因为它们是最接近的整数美元值。惟一的设计决策是对位于两个可能结果中间的数值的舍入。向偶数舍入方式采用的方法是：它将数字向上或者向下舍入，使得结果的最低有效数字是偶数。因此，这种方法将

1.5 美元和 2.5 美元都舍入成 2 美元。

方　式	\$1.40	\$1.60	\$1.50	\$2.50	\$-1.50
向偶数舍入	\$1	\$2	\$2	\$2	\$-2
向零舍入	\$1	\$1	\$1	\$2	\$-1
向下舍入	\$1	\$1	\$1	\$2	\$-2
向上舍入	\$2	\$2	\$2	\$3	\$-1

图 2.25　舍入方式说明

第一种方法舍入到一个最接近的值，而其他三种方法向上或向下限定结果。

其他三种方式产生实际值的确界（guaranteed bound）。这些方法在一些数字应用中是很有用的。向零舍入方式把正数向下舍入，把负数向上舍入，得到值 $\hat{x}$，使得$|\hat{x}| \le |x|$。向下舍入方式把正数和负数都向下舍入，得到值 x^-，使得 $x^- \le x$。向上舍入方式把正数和负数都向上舍入，得到值 x^+，满足 $x \le x^+$。

向偶数舍入初看上去好像是个相当随意的目标——有什么理由偏向取偶数呢？为什么不始终把位于两个可表示的值中间的值都向上舍入呢？使用这种方法的一个问题就是很容易假想到这样的情景：这种方法舍入一组数值，会在计算这些值的平均数中引入统计偏差。我们采用这种方式舍入得到的一组数的平均值将比这些数本身的平均值略高一些。相反，如果我们总是把两个可表示值中间的数字向下舍入，那么舍入出的一组数的平均值将比这些数本身的平均值略低一些。向偶数舍入在大多数现实情况中避免了这种统计偏差。在 50%的时间里，它将向上舍入，而在 50%的时间里，它将向下舍入。

甚至在我们不想舍入到整数时，也可以使用向偶数舍入。我们只是简单地考虑最低有效数字是奇数还是偶数。例如，假设我们想将十进制数舍入到最接近的百分位。不管用那种舍入方式，我们都将把 1.2349999 舍入到 1.23，而将 1.2350001 舍入到 1.24，因为它们不是在 1.23 和 1.24 的中间。另一方面我们将把两个数 1.2350000 和 1.2450000 都舍入到 1.24，因为 4 是偶数。

相似地，向偶数舍入法能够运用在二进制小数上。我们将最低有效位的值为 0 认为是偶数，1 认为是奇数。一般来说，只有对形如 $XX\cdots X.YY\cdots Y100\cdots$的二进制位模式的数，这种舍入方式才有效，其中 X 和 Y 表示任意位值，最右边的 Y 是要被舍入的位置。只有这种位模式表示在两个可能的值中间的值。例如，考虑舍入值到最近的四分之一的问题（也就是，二进制小数点的右两位）。我们将 10.00011_2 ($2\frac{3}{32}$)向下舍入到 10.00_2 (2)，10.00110_2($2\frac{3}{16}$)向上舍入到 10.01_2($2\frac{1}{4}$)，因为这些值不是两个可能值的中间值。我们将 10.11100_2($2\frac{7}{8}$)向上舍入成 11.00_2(3)，而 10.10100_2 向下舍入成 10.10_2($2\frac{1}{2}$)，因为这些值是两个可能值的中间值，并且我们倾向于使最低有效位为零。

2.4.5 浮点运算

IEEE 标准为诸如加法和乘法这样的算术运算的结果定义了简单的规则。把浮点值 x 和 y 看成实数，而某个运算$\odot$定义在实数上，计算将产生 Round ($x \odot y$)，这是实际运算的精确结果进行舍入后的结果。在实际中，浮点单元的设计者使用一些聪明的小技巧来避免执行这种精确的计算，因为计

算只要精确到能够保证得到一个正确舍入的结果就可以了。当参数中的一个是特殊值，如-0、-∞或*NaN*时，IEEE标准定义了一些使之更合理的规则。例如，1/-0被定义为产生-∞，而1/+0被定义为产生+∞。

IEEE标准中指定浮点运算行为的方法的一个优点在于，它可以独立于任何具体的硬件或者软件实现。因此，我们可以检查它的抽象数学属性，而不必考虑它实际上是如何实现的。

前面我们看到了整数加法（无符号和二进制补码），形成了阿贝尔群。实数上的加法也形成了阿贝尔群，但是我们必须考虑舍入对这些属性的影响。我们定义 $x +^f y$ 为 Round(x+y)。这个操作的定义针对x和y的所有取值，尽管由于溢出可能形成无穷值，即使x和y都是实数。对于所有 x 和 y 的值，这个运算是可交换的，也就是说 $x +^f y = y +^f x$。另一方面，这个运算是不可结合的。例如，使用单精度浮点，表达式(3.14+1e10)-1e10求值得到0.0——因为舍入，值3.14会丢失。另一方面，表达式3.14+(1e10-1e10)得出值3.14。作为阿贝尔群，大多数值在浮点加法下都有逆元，也就是说 $x +^f -x = 0$。例外情况是无穷（因为+∞-∞ = *NaN*）和*NaN*，因为对于任何 x，都有 $NaN +^f x = NaN$。

浮点加法不具有结合性，这是缺少的最重要的群属性。对于科学程序员和编译器编写者来说，这具有重要的含义。例如，假设一个编译器给定了如下代码段：

```
x = a + b + c;
y = b + c + d;
```

编译器可能试图通过产生下列代码来省去一个浮点加法：

```
t = b + c;
x = a + t;
y = t + d;
```

然而，对于 x 来说，这个计算可能会产生不同于原始值的值，因为它使用了加法运算的不同的结合方式。在大多数应用中，这种差异非常细小，不会太重要。不幸的是，编译器没有办法知道在效率和忠实于原始程序的确切行为之间，使用者希望达到什么样的一种平衡。结果是，它们倾向于非常保守，避免任何会对功能产生影响的优化，即使是很轻微的影响。

另一方面，浮点加法满足了下面的单调性属性：如果 $a \geq b$，那么对于任何 a 和 b 的值，除了 x 不等于 *NaN*，都有 $x + a \geq x + b$。这个实数（以及整数）加法的属性不被无符号或二进制补码加法所遵守。

浮点乘法也遵循通常乘法所具有的许多属性，也就是环的属性。我们定义 $x *^f y$ 为 Round ($x \times y$)。这个运算在乘法中是封闭的（虽然可能产生无穷大或 *NaN*），它是可交换的，而且它的乘法单位元为1.0。另一方面，由于可能发生溢出，或者由于舍入而失去精度，它不具有可结合性。例如，单精度浮点情况下，表达式(1e20*1e20)*1e－20求值为+∞，而1e20*(1e20*1e－20)将得出1e20。另外，浮点乘法在加法上不具备分配性。例如，单精度浮点情况下，表达式1e20*(1e20－1e20)求值为0.0，而1e20*1e20－1e20*1e20会得出*NaN*。

另一方面，对于任何 a、b 和 c，并且 a、b 和 c 都不等于 *NaN*，浮点乘法满足下列单调性：

$$a \geq b \text{ 且 } c \geq 0 \Rightarrow a *^f c \geq b *^f c$$

$$a \geq b \text{ 且 } c \leq 0 \Rightarrow a *^f c \leq b *^f c$$

此外，我们还可以保证，只要 $a \neq NaN$，就有 $a *^f a \geq 0$。像我们先前所看到的，无符号或二进制补码的乘法没有这些单调性属性。

对于科学程序员和编译器作者来说，缺乏结合性和分配性是很严重的问题。甚至就像写代码以确定在三维空间中两条线是否交叉这样一个看上去很简单的任务，也可能成为一个很大的挑战。

2.4.6 C 语言中的浮点

C 提供了两种不同的浮点数据类型：float 和 double。在支持 IEEE 浮点格式的机器上，这些数据类型就对应于单精度和双精度浮点。另外，这类机器使用向偶数舍入的舍入方式。不幸的是，因为 C 标准不要求机器使用 IEEE 浮点，所以没有标准的方法来改变舍入方式或者得到诸如-0、+∞、-∞或者 *NaN* 之类的特殊值。大多数系统提供 include（".h"）文件和读取这些特征的过程库，但是细节随系统不同而不同。例如，当程序文件中出现下列句子时，GNU 编译器 GCC 会定义宏 INFINITY（表示+∞）和 NAN（表示 *NaN*）：

```
# define _GNU_SOURCE 1
# include <math.h>
```

练习题 2.36

完成下列宏定义，生成双精度值+∞、-∞和 0。

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
#endif
```

不能使用任何 include 文件（例如 math.h），但你能利用这样一个事实：能够表示成双精度的最大的有限数，大约是 1.8×10^{308}。

当在 int、float 和 double 格式之间进行强制类型转换时，程序按照如下原则来转换数值和位模式（假设 int 是 32 位的）：

- 从 int 转换成 float，数字不会溢出，但是可能被舍入。
- 从 int 或 float 转换成 double，因为 double 有更大的范围（也就是可表示值的范围），也有更高的精度（也就是有效位数），所以能够保留精确的数值。
- 从 double 转换成 float，因为范围要小一些，所以值可能溢出成+∞或-∞。另外，由于精确度较小，它还可能被舍入。
- 从 float 或者 double 转换成 int，值将会向 0 截断。例如，1.999 将被转换成 1，而-1.999 将被转换成-1。注意这种行为与舍入是非常不同的。进一步来说，值可能会溢出。C 标准没有对这种情况指定固定的结果，但是在大部分机器上，结果将是 $TMax_w$ 或 $TMin_w$，其中 w 是 int 中的位数。

***Intel IA32 浮点运算**

在下一章中，我们将深入研究 Intel IA32 处理器，这种处理器大量地应用于今天的个人计算机中。这里我们重点突出这种机器的一个特性，用 GCC 编译的时候，它能够严重影响程序对浮点数运算的行为。

像大多数其他处理器一样，IA32 处理器有特别的存储器元素，称为寄存器，当计算或者使用浮点数时，用来保存浮点值。比起保存在主存中的值，保存在寄存器中的值读写起来更快。IA32 非同

一般的属性是，浮点寄存器使用一种特殊的80位的扩展精度格式，这样就比保存在存储器中的值所使用的普通32位单精度和64位双精度格式，提供了更大的表示范围和更高的精度。和在家庭作业2.58中描述的一样，扩展精度表示类似于具有15位指数（也就是$k=15$）和63位小数（也就是$n=63$）的IEEE浮点格式。所有的单精度和双精度数在从存储器加载到浮点寄存器中时，都会转换成这种格式。运算总是以扩展精度格式进行的。当数字存储在存储器中时，它们就从扩展精度转换成单精度或者双精度格式。

对于程序员而言，这种把所有寄存器数据扩展成80位，并把所有存储器数据收缩成更小的格式，会产生一些不太好的结果。这意味着在存储器中保存一个值，然后取出它就会由于舍入、下溢或者上溢，改变它的值。对于C程序员来说，这种存入和取出并不总是可见的，会导致一些奇特的结果。

下面的示例说明了这个性质：

code/data/fcomp.c

```
1   double recip(int denom)
2   {
3   return 1.0/(double) denom;
4   }
5
6   void do_nothing() {} /* Just like the name says */
7
8   void test1(int denom)
9   {
10    double r1, r2;
11    int t1, t2;
12
13    r1 = recip(denom);   /* Stored in memory */
14    r2 = recip(denom);   /* Stored in register */
15    t1 = r1 == r2;       /* Compares register to memory */
16    do_nothing();        /* Forces register save to memory */
17    t2 = r1 == r2;       /* Compares memory to memory */
18    printf("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
19    printf("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
20  }
```

code/data/fcomp.c

变量r1和r2是由有相同参数的相同函数计算的。我们会预计它们是相同的。而且，变量t1和t2都是通过对表达式r1==r2求值计算出来的，所以我们预计它们都等于1。没有明显的隐藏的副作用——函数recip进行直接的倒数计算，而且函数do_nothing就像它的名字表明的那样，什么都没干。然而，当带优化选项“-O2”编译，并用参数10运行这个文件时，我们得到下列结果：

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 == r2 0.100000
```

第一个测试表明两个倒数是不同的，而第二个测试又说它们是相同的！这当然不是我们预想的，也不是我们想要的。理解这个例子的全部细节需要我们学习GCC产生的机器级代码（参见3.14节），但是代码中的注释提供了为什么会出现这样结果的线索。函数recip计算的数值返回结果到浮点寄存器中。无论何时过程test1调用某个函数，它必须将浮点寄存器中的当前值存储到主程序栈中，这

是存放函数局部变量的地方。在执行这个存储的过程中，处理器将扩展精度寄存器值转换成双精度存储器值。因此，在第二次调用 recip（第 14 行）之前，变量 r1 被转换并存储成双精度数了。在第二个调用之后，变量 r2 有函数返回的扩展精度值。在计算 t1 时（第 15 行），双精度数 r1 与扩展精度数 r2 相比较。因为 0.1 不能精确地被任何一种格式表示，所以测试的结果是假。在调用函数 do_nothing（第 16 行）之前，r2 被转换并且存储成双精度数。在计算 t2 时（第 17 行），比较的是两个双精度数，得到结果为真。

这个示例证明了在 IA32 机器上 GCC 的一个缺陷（在 Linux 和 Microsoft Windows 系统上也有相同的结果）。由于对程序员来说不可见的运算，例如浮点寄存器的保存和恢复，变量的值发生了改变。我们对 Microsoft Visual C++编译器的测试表明它没有这种问题。

旁注：我们为什么要关心这些不一致？

正如我们将在第 5 章中讨论的，优化编译器的基本原则之一是，无论优化与否，程序应该产生完全相同的结果。不幸的是，GCC 对 IA32 机器上的浮点代码没有满足这一要求。

有一些方法来解决这个问题，不过都不是很理想。最简单的方法就是用命令行选项“-ffloat-store”来调用 GCC，告诉 GCC 每一个浮点计算的结果在使用之前都必须存储到存储器中，再读回来。这将迫使每个被计算出来的值都被转换成较低精度的形式。这样做会使程序变慢一些，但是使行为变得更加可预知。不幸的是，我们已经发现即使在给出命令行选项的情况下，GCC 也没有严格遵从先写后读的约定。例如，考虑下面的函数：

code/data/fcomp.c

```
1 void test2(int denom)
2 {
3 double r1;
4 int t1;
5 r1 = recip(denom);                   /* Default: register, Forced store: memory */
6 t1 = r1 == 1.0/(double) denom;       /* Compares register or memory to register */
7 printf("test2 t1: r1 %f %c= 1.0/10.0\n", r1, t1 ? '=' : '!');
8 }
```

code/data/fcomp.c

当只带“-O2”选项编译时，t1 得到值 1——比较是在两个寄存器值之间进行的。当带“-ffloat-store”选项编译时，t1 得到值 0！虽然函数 recip 调用的结果被写入存储器，并且读回到一个寄存器中，然而，`1.0/(double) denom` 计算出的值是保存在寄存器中的。总地来说，我们发现程序中看起来微小的改变能够引起这些测试以不可预知的方式成功或者失败。

另外一种选择是，我们能够通过将所有的变量声明为 long double 类型，而让 GCC 在所有的计算中都使用扩展精度，如下面的代码段所示：

code/data/fcomp.c

```
1 long double recip_l(int denom)
2 {
3 return 1.0/(long double) denom;
4 }
5
```

```
6 void test3(int denom)
7 {
8 long double r1, r2;
9 int t1, t2, t3;
10
11 r1 = recip_l(denom);    /* Stored in memory */
12 r2 = recip_l(denom);    /* Stored in register */
13 t1 = r1 == r2;          /* Compares register to memory */
14 do_nothing();           /* Forces register save to memory */
15 t2 = r1 == r2;          /* Compares memory to memory */
16 t3 = r1 == 1.0/(long double) denom; /* Compare memory to register */
17 printf("test3 t1: r1 %f %c= r2 %f\n",
18 (double) r1, t1 ? '=' : '!', (double) r2);
19 printf("test3 t2: r1 %f %c= r2 %f\n",
20 (double) r1, t2 ? '=' : '!', (double) r2);
21 printf("test3 t3: r1 %f %c= 1.0/10.0\n",
22 (double) r1, t2 ? '=' : '!');
23 }
```

code/data/fcomp.c

ANSI C 标准允许 long double 类型的声明，虽然对于大多数机器和编译器而言，这个声明等价于普通的 double 类型。然而，对于 IA32 机器上的 GCC 来说，它会对存储器数据使用扩展精度格式，就像对浮点寄存器数据一样。这就使得我们能够充分利用扩展精度格式提供的更广的范围和更大的精度，从而避免我们在先前的例子中看到的异常现象。不幸的是，这种解决方式是要付出代价的。GCC 使用 12 字节来存储 long double 类型，增加了 50%的存储器消耗（虽然 10 个字节已经足够了，但是使用 12 字节能获得更好的存储器性能。Linux 和 Windows 机器上使用相同的分配方式）。在寄存器和存储器之间传送这些更长的数据也需要更多的时间。尽管如此，这仍然是程序想要得到最准确和可预知结果的最好选择。

旁注：Ariane 5——浮点溢出的高昂代价

将大的浮点数转换成整数是一种常见的程序错误来源。1996 年 6 月 4 日，对于 Ariane 5 火箭的初次航行来说，这样一个错误产生了灾难性的后果。发射后仅仅 37 秒钟，火箭偏离了它的飞行路径，解体并且爆炸了。火箭上载有价值 5 亿美元的通信卫星。

后来的调查[49]显示，控制惯性导航系统的计算机向控制引擎喷嘴的计算机发送了一个无效数据。它没有发送飞行控制信息，而是送出了一个诊断位模式，表明在将一个 64 位浮点数转换成 16 位有符号整数时，产生了溢出。

溢出值测量的是火箭的水平速率，这比早先的 Ariane 4 火箭所能达到的高出了 5 倍。在设计 Ariane 4 火箭的软件时，他们小心地分析了数字值，并且确定水平速率决不会超出一个 16 位的数。不幸的是，他们在 Ariane 5 火箭的系统中简单地重新使用了这一部分，而没有检查它所基于的假设。

练习题 2.37

假定变量 x、f 和 d 的类型分别是 int、float 和 double。它们的值是任意的，除了 f 和 d 都不能

等于+∞、-∞或者 *NaN*。对于下面每个 C 表达式，要么证明它总是为真（也就是，求值为 1），或者给出一个使表达式不为真的值（也就是，求值为 0）。

```
A. x == (int)(float) x
B. x == (int)(double) x
C. f == (float)(double) f
D. d == (float) d
E. f == -(-f)
F. 2/3 == 2/3.0
G. (d >= 0.0) || ((d*2) < 0.0)
H. (d+f)-d == f
```

2.5 小结

计算机将信息编码为位（比特），通常组织成字节序列。有不同的编码方式用来表示整数、实数和字符串。不同的计算机模型在编码数字和多字节数据中的字节顺序上使用不同的约定。

C 语言被设计成包容多种不同字长和数字编码的实现。虽然高端机器逐渐开始使用 64 位字长，但是目前大多数机器仍使用 32 位字长。大多数机器对整数使用二进制补码编码，而对浮点数使用 IEEE 编码。在位级上理解这些编码，并且理解算术运算的数学特性，对于编写能在全部数值范围上正确运算的程序来说，是很重要的。

C 语言的标准规定在无符号和有符号整数之间进行强制类型转换时，基本的位模式不应该改变。在二进制补码机器上，对于一个 w 位的值，这种行为是由函数 $T2U_w$ 和 $U2T_w$ 来描述的。C 语言隐式的强制类型转换会得到许多程序员无法预计的结果，常常导致程序错误。

由于编码的长度有限，计算机运算与传统整数和实数运算相比，具有非常不同的属性。当超出表示范围时，有限长度能够引起数值溢出。当浮点数非常接近于 0.0，从而转换成零时，浮点数也会下溢。

和大多数其他程序语言一样，C 语言实现的有限整数运算和真实的整数运算相比有一些特殊的属性。例如，由于溢出，表达式 $x*x$ 能够得出负数。但是，无符号数和二进制补码的运算都满足环的属性。这就允许编译器做很多的优化。例如，用$(x<<3)-x$ 取代表达式 $7*x$ 时，我们就利用了结合性、交换性和分配性，还利用了移位和乘以 2 的幂之间的关系。

我们已经看到了几种使用位级运算和算术运算组合的聪明方法。例如，我们看到，使用二进制补码运算，$\sim x+1$ 是等价于 $-x$ 的。另外一个例子，假设我们想要一个形如[0,⋯,0,1,⋯,1]的位模式，由 $w-k$ 个 0 后面紧跟着 k 个 1 组成。这些位模式对于掩码运算是很有用的。这种模式能够通过 C 表达式$(1<<k)-1$ 生成，利用的是这样一个属性，即我们想要的位模式的数值为 2^k-1。例如，表达式(1<<8) -1 将产生位模式 0xFF。

浮点表示通过将数字编码为 $x\times 2^y$ 的形式来近似地表示实数。最常见的浮点表示方式是由 IEEE 标准 754 定义的。它提供了几种不同的精度，最常见的是单精度（32 位）和双精度（64 位）。IEEE 浮点也能够表示特殊值∞和 NaN。

必须非常小心地使用浮点运算，因为浮点运算的范围和精度有限，而且浮点运算并不遵守普遍的算术属性，比如结合性。

参考文献说明

关于C的参考书[40，32]讨论了不同的数据类型和运算的属性。C标准对于精确的字长或者数字编码没有详细的定义。这些细节是故意省去的，以使得可以在更大范围的不同机器上实现C语言。已经有几本书[41，50]给了C语言程序员一些建议，警告他们关于溢出、隐含强制类型转换到无符号数，以及其他一些我们已经在这一章中谈及到的陷阱。这些书还提供了对变量命名、编码风格和代码测试的有益建议。关于Java的书（我们推荐Java语言的创始人James Gosling参与编写的一本书[1]）描述了Java支持的数据格式和算术运算。

大多数关于逻辑设计的书[86，39]都有关于编码和算术运算的章节。这些书描述了实现算术电路的不同方式。Overton的关于IEEE浮点的书[56]提供了从一个数字应用程序员的角度出发的关于格式和属性的详细描述。

家庭作业

◆ = 考验概念的快速习题

◆◆ = 需要5～15分钟来完成，可能包括编写和运行程序

◆◆◆ = 需要几个小时来完成的持续习题

◆◆◆◆ = 需要一个或者两个星期来完成的实验任务

2.38 ◆

在你能够访问的不同机器上，编译并运行使用show_bytes的示例代码（文件show-bytes.c）。确定这些机器使用的字节顺序。

2.39 ◆

试着用不同的示例值来运行show_bytes的代码。

2.40 ◆

编写程序show_short、show_long和show_double，它们分别打印类型为short int、long int和double的C语言对象的字节表示。请在几种机器上运行。

2.41 ◆◆

编写过程is_little_endian，当在小端法机器上编译和运行时返回1，在大端法机器上编译运行时则返回0。这个程序应该可以运行在任何机器上，无论机器的字长是多少。

2.42 ◆◆

编写一个C表达式，它生成一个字，由x的最低有效字节和y中剩下的字节组成。对于运算数x = 0x89ABCDEF 和 y = 0x76543210，就得到 0x765432EF。

2.43 ◆◆

只使用位级和逻辑运算，编写出C的表达式，在下列描述的条件下产生1，而在其他情况下得到0。你的代码应该能工作在任何字长的机器上。假设x是整数。

A. x的任何位都等于1。

B. x的任何位都等于0。

C. x的最低有效字节中的位都等于1。

D. x 的最低有效字节中的位都等于 0。

2.44 ◆◆◆

编写一个函数 int_shifts_are_arithmetic()，使得这个函数在对整数使用算术右移的机器上运行时生成 1，而其他情况下生成 0。你的代码应该可以运行在任何字长的机器上。在几种机器上测试你的代码。编写并测试过程 unsigned_shifts_are_arithmetic()，该过程确定对无符号整数使用的移位形式。

2.45 ◆◆

你有一个任务，要编写一个过程 int_size_is_32()，当在一个 int 是 32 位的机器上运行时，该程序产生 1，而对于其他情况则生成 0。下面是开始时的尝试：

```
1  /* The following code does not run properly on some machines */
2  int bad_int_size_is_32()
3  {
4      /* Set most significant bit (msb) of 32-bit machine */
5      int set_msb = 1 << 31;
6      /* Shift past msb of 32-bit word */
7      int beyond_msb = 1 << 32;
8
9      /* set_msb is nonzero when word size >= 32
10        beyond_msb is zero when word size <= 32 */
11     return set_msb && !beyond_msb;
12 }
```

不过，当在 SUN SPARC 这样的 32 位机器上编译并运行时，这个过程返回的却是 0。下面的编译器信息给了我们一个问题的指示：

```
warning: left shift count >= width of type
```

A. 我们的代码在哪个方面没有遵守 C 的标准？

B. 修改代码，使得它在 int 至少为 32 位的任何机器上都能正确地运行。

C. 修改代码，使得它在 int 至少为 16 位的任何机器上都能正确地运行。

2.46 ◆

你刚刚开始为一家公司工作，他们要实现一组过程来操作一个数据结构，要将 4 个有符号字节封装成一个 32 位 unsigned。在字中的字节是从 0（最低有效字节）编号到 3（最高有效字节）。你被分配的任务是：为使用二进制补码运算和算术右移的机器编写一个具有如下原型的函数：

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word. Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

也就是说，函数会抽取出指定的字节，再把它符号扩展为一个 32 位 int。

你的前任（因为水平不够高而被解雇了）编写了下面的代码：

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
```

```
{
    return
        (word >> (bytenum << 3)) & 0xFF;
}
```

A. 这段代码错在哪里？

B. 给出函数的正确实现，只使用左右移位和一个减法。

2.47 ◆

填写下列表格，按照图 2.20 的风格，表明对五位向量取补（或取反）和加 1 的结果。请展示位向量和数值。

$\vec{x}$	$\sim\vec{x}$	$incr(\sim\vec{x})$
[01101]		
[01111]		
[11000]		
[11111]		
[10000]		

2.48 ◆◆

请说明先减 1 然后取补等价于先取补然后再加 1。也就是说，对于任意有符号值 x，C 表达式-x、~x+1 和~(x-1)产生同样的结果。你的推导依赖于二进制补码加法的什么数学属性？

2.49◆◆◆

假设我们想要计算 $x \cdot y$ 的完全 $2w$ 位表示，其中，x 和 y 都是无符号数，并且运行的机器上数据类型 unsigned 是 w 位的。乘积的低 w 位能够用表达式 x * y 计算，所以，我们只需要一个具有下列原型的函数

```
unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

这个函数计算无符号变量 $x \cdot y$ 的高 w 位。

我们使用一个具有下面原型的库函数：

```
int signed_high_prod(int x, int y);
```

它计算在 x 和 y 是二进制补码形式的情况下，$x \cdot y$ 的高 w 位。编写代码调用这个过程，以实现用无符号数为参数的函数。验证你的解答的正确性。

提示：看看等式 2.16 的推导中，有符号乘积 $x \cdot y$ 和无符号乘积 $x' \cdot y'$ 之间的关系。

2.50 ◆◆

假设我们有一个任务：生成一段代码，用来将整数变量 x 乘以不同的常数因子 K。为了提高效率，我们想只使用+、-、<<等运算。对于下列 K 的值，写出执行乘法运算的 C 表达式，每个表达式中最多使用 3 个运算。

A. K=5：

B. K=9：

C. K=14：

D. K=-56;

2.51 ◆◆

编写产生如下位模式的 C 表达式，其中 a^k 表示符号 a 重复 k 次。假设一个 w 位的数据类型。你的代码可以包含对参数 j 和 k 的引用，它们分别表示 j 和 k 的值，但是不能使用表示 w 的参数。

A. $1^{w-k}0^k$。

B. $0^{w-k-j}1^k0^j$。

2.52 ◆◆

假设我们把 w 位的字中的字节按照从 0（最低有效字节）到 w/8-1（最高有效字节）的顺序编号。写出下面 C 函数的代码，这段代码将返回一个无符号值，其中参数 x 的字节 i 已经被字节 b 覆盖了：

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

下面是一些示例，展示了函数会如何工作：

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

2.53 ◆◆◆

填写下列 C 函数的代码。函数 srl 使用算术右移（由值 xsra 给出）来执行逻辑右移，紧跟着的是其他不包括右移或者除法的运算。函数 sra 使用逻辑右移（由值 xsrl 给出）来执行算术右移，紧跟着的是其他不包括右移或者除法的运算。你可以假设 int 是 32 位长的，移位量 k 的取值范围是 0~31。

```
unsigned srl(unsigned x, int k)
{
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;

    /* ... */
}
int sra(int x, int k)
{
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;

    /* ... */
}
```

2.54 ◆

我们在一个 int 类型值为 32 位的机器上运行程序。这些值以二进制补码表示，而且它们都是算术右移的。unsigned 类型的值也是 32 位的。

我们产生任意值 x 和 y，并且把它们转换成其他无符号数：

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
```

```
unsigned uy = (unsigned) y;
```

对于下列每个 C 表达式，你要指出表达式是否总是为 1。如果它总是为 1，那么请描述其中的数学原理。否则，列举出一个使它为 0 的参数示例。

A. `(x<y) == (-x>-y)`

B. `((x+y)<<4) + y-x == 17*y+15*x`

C. `~x+~y == ~(x+y)`

D. `(int) (ux-uy) == -(y-x)`

E. `((x >> 1) << 1) <= x`

2.55 ◆◆

考虑这样一些数字，它们的二进制表示是由形如 0.y y y y y y …的无穷串组成的，其中 y 是一个 k 位的序列。例如，$\frac{1}{3}$ 的二进制表示是 0.01010101…($y=01$)，而 $\frac{1}{5}$ 的二进制表示是 0.001100110011…($y=0011$)。

A. 设 $Y=B2U_k(y)$，也就是说，这个数具有二进制表示 y。对于无穷串表示的值，给出一个由 Y 和 k 组成的公式。

提示：请考虑将二进制小数点右移 k 位的结果。

B. 对于下列的 y 值，串的数值是多少？

(a) 001

(b) 1001

(c) 000111

2.56 ◆

填写下列程序的返回值，这个程序测试的是它的第一个参数是否大于或者等于第二个参数。假定函数 f2u 返回一个无符号 32 位数字，其位表示与它的浮点参数相同。你可以假设参数都不是 NaN。+0 和-0 被认为是相等的。

```
int float_ge(float x, float y)
{
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return /* ... */ ;
}
```

2.57 ◆

给定一个浮点格式，有 k 位指数和 n 位小数，对于下列数，写出指数 E、有效数 M、小数 f 和值 V 的公式。另外，请描述其位表示。

A. 数 5.0。

B. 能够被准确描述的最大奇整数。

C. 最小的规格化数的倒数。

2.58 ◆

与 Intel 兼容的处理器也支持“扩展精度”浮点形式，这种格式具有 80 位字长，被分成 1 个符号位、15 个指数位（k=15）、1 个单独的整数位和 63 个小数位（n=63）。整数位是 IEEE 浮点表示中隐含位的显式拷贝。也就是说，对于标准值它等于 1，对于不标准值它等于 0。填写下表，给出这种格式中的一些“有趣的”数字的近似值。

描 述	扩展精度	
	值	十进制
最小的非规格化数		
最小的规格化数		
最大的规格化数		

2.59 ◆

考虑一个基于 IEEE 浮点格式的 16 位浮点表示，它具有 1 个符号位、7 个指数位（k=7）和 8 个小数位（n=8）。指数偏置量是 $2^{7-1}-1=63$。

对于每个给定的数，填写下表，其中，每一列具有如下指示说明：

Hex：描述编码形式的四个十六进制数字。

M：有效数的值。这应该是一个形如 x 或 $\frac{x}{y}$ 的数，其中 x 是一个整数，而 y 是 2 的整数幂。例如：0、$\frac{67}{64}$ 和 $\frac{1}{256}$。

E：指数的整数值。

V：所表示的数字值。使用 x 或者 $x\times 2^z$ 表示，其中 x 和 z 都是整数。

举一个例子，为了表示数 $\frac{7}{2}$，我们有 $s=0$，$M=\frac{7}{4}$ 和 E=1。因此我们的数的指数域为 0x40（十进制值 63+1=64），有效数域为 0xC0（二进制 11000000_2），得到一个十六进制的表示 40C0。

标记为“-”的条目不用填写。

描 述	Hex	M	E	V
−0				—
最小>1 的值				
256				—
最大的非规格化数				
−∞		—	—	—
十六进制表示为 3AA0 的数	—			

2.60 ◆

我们在一个 int 类型为 32 位二进制补码表示的机器上运行程序。float 类型的值使用 32 位 IEEE 格式，而 double 类型的值使用 64 位 IEEE 格式。

我们产生任意的整数值 x、y 和 z，并且把它们转换成 double：

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

对于下列的每个 C 表达式，你要指出表达式是否总是为 1。如果它总是为 1，描述其中的数学原理。否则，列举出使它为 0 的参数的例子。请注意，不能使用 IA32 机器运行 GCC 来测试你的答案，因为对于 float 和 double，它使用的都是 80 位的扩展精度表示。

A. `(double)(float) x == dx`

B. `dx + dy == (double) (y+x)`

C. `dx + dy + dz == dz + dy + dx`

D. `dx * dy * dz == dz * dy * dx`

E. `dx / dx == dy / dy`

2.61 ◆

你被分配了一个任务，要编写一个 C 函数来计算 2^x 的浮点表示。你意识到完成这个的最好方法是直接创建结果的 IEEE 单精度表示。当 x 太小时，你的程序将返回 0.0。当 x 太大时，它会返回 $+\infty$。填写下列代码的空白部分，以计算出正确的结果。假设函数 u2f 返回的浮点值与它的无符号参数有相同的位表示。

```
float fpwr2(int x)
{
    /* Result exponent and significand */
    unsigned exp, sig;
    unsigned u;

    if (x < ______)           /* Too small. Return 0.0 */
        exp = ____________;
        sig = ____________;
    } else if (x < ______)    /* Denormalized result */
        exp = ____________;
        sig = ____________;
    } else if (x < ______)    /* Normalized result. */
```

```
        exp = ____________;
        sig = ____________;
    } else {                    /* Too big. Return +oo */
        exp = ____________;
        sig = ____________;

    /* Pack exp and sig into 32 bits */
    u = exp << 23 | sig;
    /* Return as float */
    return u2f(u);
}
```

2.62 ◆

大约公元前250年，希腊数学家阿基米德证明了$\frac{223}{71}<\pi<\frac{22}{7}$。如果当时他有一台计算机和标准库<math.h>，他就能够确定π的单精度浮点近似值的十六进制表示为0x40490FDB。当然，所有的这些都只是近似值，因为π不是有理数。

A. 这个浮点值表示的二进制小数是多少？

B. $\frac{22}{7}$的二进制小数表示是什么？**提示**：参见练习题2.55。

C. 这两个π的近似值从哪一位（相对于二进制小数点）开始不同的？

练习题答案

练习题 2.1 答案

一旦我们开始查看机器级程序，理解十六进制和二进制格式的关系将是很重要的。虽然本书中介绍了完成这些转换的方法，但是做点练习能够让你更加熟练。

A. 将0x8F7A93转换成二进制：

十六进制	8	F	7	A	9	3
二进制	1000	1111	0111	1010	1001	0011

B. 将二进制1011011110011100转换成十六进制：

二进制	1011	0111	1001	1100
十六进制	B	7	9	C

C. 将0xC4E5D 转换成二进制：

十六进制	C	4	E	5	D
二进制	1100	0100	1110	0101	1101

D. 将二进制1101011011011111100110转换成十六进制：

二进制	11	0101	0111	1110	0110
十六进制	3	5	7	E	6

练习题 2.2 答案

这个问题给你一个机会思考 2 的幂和它们的十六进制表示。

n	2^n（十进制）	2^n（十六进制）
11	2048	0x800
7	128	0x80
13	8192	0x2000
17	131072	0x20000
16	66536	0x10000
8	256	0x100
5	32	0x20

练习题 2.3 答案

这个问题给你一个机会试着对一些小的数在十六进制和十进制表示之间进行转换。对于较大的数，使用计算器或者转换程序会更加方便和可靠一些。

十进制	二进制	十六进制
0	00000000	00
55=3·16+7	0011 0111	37
136=8·16+8	1000 1000	88
243=15·16+3	1111 0011	F3
5·16+2=82	0101 0010	52
10·16+12=172	1010 1100	AC
14·16+7=231	1110 0111	E7
10·16+7=167	1010 0111	A7
3·16+14=62	0011 1110	3E
11·16+12=188	1011 1100	BC

练习题 2.4 答案

当开始调试机器级程序时，将发现在许多情况中，一些简单的十六进制运算是很有用的。可以总是把数转换成十进制，完成运算，再把它们转换回来，但是能够直接用十六进制工作更加有效，而且能够提供更多的信息。

A. 0x502c+0x8 = 0x5034。8 加上十六进制 c 得到 4 并且进位 1。

B. 0x502c-0x30 = 0x4ffc。在第二个数位，2 减去 3 要求从第三位借 1。因为第三位是 0，所以我们必须从第四位借位。

C. 0x502c+64 = 0x506c。十进制 64（2^6）等于十六进制 0x40。

D. 0x51da-0x502c = 0xae。十六进制数 a（十进制 10）减去十六进制数 c（十进制 12），我们从

第二位借 16，得到十六进制数 e（十进制数 14）。在第二个数位，我们现在用十六进制 c（十进制 12）减去 2，得到十六进制 a（十进制 10）。

练习题 2.5 答案

这个问题测试你对数据的字节表示和两种不同字节顺序的理解。

A. 小端法：78　　　　　　大端法：12

B. 小端法：78　56　　　　大端法：12　34

C. 小端法：78　56　34　　大端法：12　34　56

回想一下，show_bytes 列举了一系列字节，从低位地址的字节开始，然后逐一列出高位地址的字节。在一个小端法机器上，它将按照从最低有效字节到最高有效字节的顺序列出字节。在一个大端法机器上，它将按照从最高有效字节到最低有效字节的顺序列出字节。

练习题 2.6 答案

这个问题又是一个练习从十六进制到十进制转换的机会。同时它带给你对整数和浮点表示的思考。我们将在本章后面更加详细地研究这些表示。

A. 利用书中示例的符号，我们将两个串写成：

```
   0    0    3    5    4    3    2    1
00000000001101010100001100100001
            *********************
         4    A    5    5    0    C    8    4
      01001010010101010000110010000100
```

B. 将第二个字相对于第一个字移动 2 位，我们发现一个有 21 个匹配位的序列。

C. 我们发现除了最高位 1，整数的所有位都嵌入在浮点数中。这正好是书中示例的情况。另外，浮点数有一些非零的高位不与整数中的高位相匹配。

练习题 2.7 答案

它打印出 41 42 43 44 45 46。回想一下，库函数 strlen 不计算终止的空字符，所以 show_bytes 只打印到字符“F”。

练习题 2.8 答案

这个题目是一个帮助你更加熟悉布尔运算的练习。

运算	结果
a	[01101001]
b	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
$a \& b$	[01000001]
$a \mid b$	[01111101]
a ^ b	[00111100]

练习题 2.9 答案

这个问题举例说明了布尔代数怎样被用来描述和解释现实世界的系统。我们能够看到这个颜色代数和使用长度为3的位向量上的布尔代数是一样的。

A. 颜色的取补是通过对R、G和B的值取补得到的。由此，我们可以看出，白色是黑色的补，黄色是蓝色的补，红紫色是绿色的补，蓝绿色是红色的补。

B. 黑色是0，而白色是1。

C. 我们基于颜色的位向量表示来进行布尔运算。据此，我们得到以下结果：

蓝色（001）| 红色（100）= 红紫色（101）

红紫色（101）& 蓝绿色（011）= 蓝色（001）

绿色（010）^ 白色（111）= 红紫色（101）

练习题 2.10 答案

这个程序依赖于 EXCLUSIVE-OR 是可交换的和可结合的这一事实，以及对于任意的 a，有 a ^a=0。在第5章中我们将看到当两个指针 x 和 y 相等时（也就是说，两个指针指向同一个位置时），这段代码将工作得不正确。

步骤	*x	*y
初始	a	b
步骤1	a ^ b	b
步骤2	a ^ b	$(a$ ^ $b)$ ^ $b = (b$ ^ $b)$ ^ $a = a$
步骤3	$(a$ ^ $b)$ ^ $a = (a$ ^ $a)$ ^ $b = b$	a

练习题 2.11 答案

观察下列表达式：

A. x | ~0xFF

B. x ^ 0xFF

C. x & ~0xFF

这些表达式是在执行低级位运算中经常发现的典型类型。表达式~0xFF创建一个掩码，该掩码8个最低位等于0，而其余的位为1。可以观察到，这些掩码的产生是和字长无关的。而相比之下，表达式0xFFFFFF00只能工作在32位的机器上。

练习题 2.12 答案

这个问题帮助你思考布尔运算和典型的掩码运算之间的关系。代码如下：

```
/* Bit Set */
int bis(int x, int m)
{
int result = x | m;
return result;
}
/* Bit Clear */
```

```
int bic(int x, int m)
{
int result = x & ~m;
return result;
}
```

很容易看出，bis是等价于布尔OR——如果x中或者m中的这一位置位了，那么z中的这一位就置位。

bic运算更加微妙一些。我们想要设置z的一位为0，如果m的相应位等于1。若我们对这个掩码取补得到~m，那么我们就是想要设置z的一位为0，如果取补后的掩码的相应位等于0。我们能够用AND运算来实现这一点。

练习题2.13 答案

这个问题突出说明了位级布尔运算和C语言中的逻辑运算之间的关系：

表达式	值	表达式	值
x & y	0x02	x && y	0x01
x | y	0xF7	x || y	0x01
~x | ~y	0xFD	!x || !y	0x00
x & !y	0x00	x && ~y	0x01

练习题2.14 答案

表达式是 !(x^y)。

也就是，当且仅当x的每一位和y相应的每一位匹配时，$x \wedge y$等于零。然后，我们利用！的功能来判定一个字是否包含任何非零位。

没有任何实际的理由要去使用这个表达式，而不简单地写成 $x == y$，但是它说明了位级运算和逻辑运算之间的一些细微差别。

练习题2.15 答案

这个问题是一个帮助你理解不同移位运算的练习。

x		x<<3		x>>2（逻辑）		x>>2（算术）	
十六进制	二进制	二进制	十六进制	二进制	十六进制	二进制	十六进制
0xF0	[11110000]	[10000000]	0x80	[00111100]	0x3C	[11111100]	0xFC
0x0F	[00001111]	[01111000]	0x78	[00000011]	0x03	[00000011]	0x03
0xCC	[11001100]	[01100000]	0x60	[00110011]	0x33	[11110011]	0xF3
0x55	[01010101]	[10101000]	x0A8	[00010101]	0x15	[00010101]	0x15

练习题2.16 答案

一般而言，研究非常小的字长的例子是理解计算机运算的非常好的方法。

无符号值对应于图2.1中的值。对于二进制补码值，十六进制数字0~7的最高有效位为0，得

到非负值，然而十六进制数字 8~F 的最高有效位为 1，得到一个为负的值。

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
十六进制	二进制		
A	[1010]	$2^3+2^1=10$	$-2^3+2^1=-6$
0	[0000]	0	0
3	[0011]	$2^1+2^0=3$	$2^1+2^0=3$
8	[1000]	$2^3=8$	$-2^3=-8$
C	[1100]	$2^3+2^2=12$	$-2^3+2^2=-4$
F	[1111]	$2^3+2^2+2^1+2^0=15$	$-2^3+2^2+2^1+2^0=-1$

练习题 2.17 答案

对于 32 位的机器，任何值如果由 8 个十六进制数字组成的，而且开始的那个数字在 8~f 之间，那么这个数值就是一个负数。看到数字以串 f 开头是很普遍的事情，因为负数的起始位全为 1。不过，你必须看仔细了。例如，数 0x80483b7 仅仅有 7 个数字，把起始位填入 0，从而得到 0x080483b7，这是一个正数。

```
80483b7: 81 ec 84 01 00 00     sub $0x184,%esp                A. 388
80483bd: 53                    push %ebx
80483be: 8b 55 08              mov 0x8(%ebp),%edx             B. 8
80483c1: 8b 5d 0c              mov 0xc(%ebp),%ebx             C. 12
80483c4: 8b 4d 10              mov 0x10(%ebp),%ecx            D. 16
80483c7: 8b 85 94 fe ff ff     mov 0xfffffe94(%ebp),%eax      E. -364
80483cd: 01 cb                 add %ecx,%ebx
80483cf: 03 42 10              add 0x10(%edx),%eax            F. 16
80483d2: 89 85 a0 fe ff ff     mov %eax,0xfffffea0(%ebp)      G. -352
80483d8: 8b 85 10 ff ff ff     mov 0xffffff10(%ebp),%eax      H. -240
80483de: 89 42 1c              mov %eax,0x1c(%edx)            I. 28
80483e1: 89 9d 7c ff ff ff     mov %ebx,0xffffff7c(%ebp)      J. -132
80483e7: 8b 42 18              mov 0x18(%edx),%eax            K. 24
```

练习题 2.18 答案

从数学的视角来看，函数 *T2U* 和 *U2T* 是非常奇特的。理解它们的行为非常重要。

解答这个问题，我们是根据二进制补码的值，重新排列练习题 2.16 的解答中的行，然后列出无符号值作为函数应用的结果。我们展示出十六进制值，以使这个进程更加具体。

$\vec{x}$（十六进制）	x	$T2U_4(x)$
8	-8	8
A	-6	10
C	-4	12
F	-1	15
0	0	0
3	3	3

练习题 2.19 答案

这个练习题测试你对等式 2.4 的理解。

对于开始的四个条目，x 的值是负的，并且 $T2U_4(x) = x + 2^4$。对于剩下的两个条目，x 的值是非负的，并且 $T2U_4(x) = x$。

练习题 2.20 答案

这个问题加强你对二进制补码和无符号表示之间关系的理解，以及对 C 语言升级规则（promotion rule）的影响的理解。回想一下，$TMin_{32}$ 是-2147483648，并且将它强制类型转换为无符号数后，变成了 2147483648。另外，如果有任一个运算数是无符号的，那么在比较之前，另一个运算数会被强制类型转换为无符号数。

表 达 式	类 型	求值
`-2147483648 == 2147483648U`	无符号数	1
`-2147483648 < -2147483647`	有符号数	1
`(unsigned) -2147483648 < -2147483647`	无符号数	1
`-2147483648 < 2147483647`	有符号数	1
`(unsigned) -2147483648 < 2147483647`	无符号数	0

练习题 2.21 答案

在这些函数中的表达式是常见的程序“习惯用语”，用来从多个位域打包成的一个字中提取值。它们利用不同移位运算的零填充和符号扩展属性。请注意强制类型转换和移位运算的顺序。在 fun1 中，移位是在无符号 word 上进行的，因此是逻辑移位。在 fun2 中，移位是在把 word 强制类型转换为 int 之后进行的，因此是算术移位。

A.

w	fun1(w)	fun2(w)
127	127	127
128	128	-128
255	255	-1
256	0	0

B. 函数 fun1 从参数的低 8 位中提取一个值，得到范围 0～255 之间的一个整数。函数 fun2 也从这个参数的低 8 位中提取一个值，但是它还要执行符号扩展。结果将是介于-128～127 之间的一个数。

练习题 2.22 答案

对于无符号数，截断的影响是相当直观的，但是对于二进制补码数就不是这样的了。这个练习让你使用非常小的字长来研究它的属性。

正如等式 2.7 所描述的，这种截断无符号数值的结果就是发现它们的模 8 余数。截断有符号数

的结果要更复杂一些。根据等式 2.8，我们首先计算这个参数模 8 后的余数。对于参数 0~7，这将得出值 0~7，对于参数-8~-1 也是一样。然后我们对这些余数应用函数 $U2T_3$，得出两个 0~3 和-4~-1 序列的反复。

十六进制		无符号		二进制补码	
原始数	截断后的数	原始数	截断后的数	原始数	截断后的数
0	0	0	0	0	0
3	3	3	3	3	3
8	0	8	0	-8	0
A	2	10	2	-6	2
F	7	15	7	-1	-1

练习题 2.23 答案

这个问题是设计来说明从有符号数到无符号数的隐式强制类型转换是多么容易引起错误的啊。将参数 length 作为一个无符号数来传递看上去是件相当自然的事情，因为没有人会想到使用一个值为负数的 length。停止条件 i <= length-1 看上去也很自然。但是把这两点组合到一起，将产生意想不到的结果！

因为参数 length 是无符号的，计算 0-1 将使用无符号运算来进行，这相当于模数加法。结果是 $UMax_{32}$（假设是 32 位的机器）。≤比较同样使用无符号数比较，而因为任意的 32 位数都是小于或者等于 $UMax_{32}$ 的，所以这个比较运算将一直持续下去！因此，代码将试图访问数组 a 的无效元素。

有两种方法可以改正这段代码，其一是将 length 声明为 int 类型，其二是将 for 循环的测试条件改为 i < length。

练习题 2.24 答案

这道习题是对算术模 16 的简单示范。最容易的解决方法是将十六进制模式转换成它的无符号十进制值。对于非零的 x 值，我们必须有 $(-^{u}_{4}\, x) + x = 16$。然后，我们就可以将取补了的值转换回十六进制。

x		$-^{u}_{4}\, x$	
十六进制	十进制	十进制	十六进制
0	0	0	0
3	3	13	D
8	8	8	8
A	10	6	6
F	15	1	1

练习题 2.25 答案

这道习题是一个确保你理解了二进制补码加法的练习。

x	y	$x+y$	$x+^{t}_{4}y$	情况
-16 [10000]	-11 [10101]	-27	5 [00101]	1
-16 [10000]	-16 [10000]	32	0 [00000]	1
-8 [11000]	7 [00111]	-1	-1 [11111]	2
-2 [11110]	5 [00101]	3	3 [00011]	3
8 [01000]	8 [01000]	16	-16 [10000]	4

练习题 2.26 答案

这个问题使用非常小的字长来帮助你理解二进制补码的非（negation）。

对于 $w=4$，我们有 $TMin_4=-8$。因此-8 是它自己的加法逆元，而其他数值是通过整数非反来取非的。

x		$-^{u}_{4}x$	
十六进制	十进制	十进制	十六进制
0	0	0	0
3	3	-3	D
8	-8	-8	8
A	-6	6	6
F	-1	1	1

对于无符号数非，位的模式是相同的。

练习题 2.27 答案

这道习题是一个确保你理解了二进制补码乘法的练习。

模式	x		y		$x\cdot y$		截断了的 $x\cdot y$	
无符号数	6	[110]	2	[010]	12	[001100]	4	[100]
二进制补码	2	[110]	2	[010]	-4	[111100]	-4	[100]
无符号数	1	[001]	7	[111]	7	[000111]	7	[111]
二进制补码	1	[001]	-1	[111]	-1	[111111]	7	[111]
无符号数	7	[111]	7	[111]	49	[110001]	1	[001]
二进制补码	-1	[111]	1	[111]	1	[000001]	1	[001]

练习题 2.28 答案

在第 3 章中，我们将看到很多实际的 leal 指令的例子。这个指令被提供用来支持指针运算，但

是 C 编译器经常用它来作为执行小常数乘法的一种方法。

对于 k 的每一个值，我们可以计算出 2 的倍数：2^k（当 b 为 0）和 2^k+1（当 b 为 a）。因此我们能够计算出倍数为 1, 2, 3, 4, 5, 8 和 9。

练习题 2.29 答案

我们发现当人们直接用汇编代码做这个练习时是有困难的。但当把它放入到 optarith 所示的形式中，问题就变得更加清晰明了了。

我们可以看到 M 是 15；$x*M$ 是作为 $(x<<4)-x$ 来计算。

我们可以看到 N 是 4；当 y 是负数时，加上偏置量 3，并且右移 2 位。

练习题 2.30 答案

这个"C 的迷题"清楚地告诉程序员必须理解计算机运算的属性。

A. (x >= 0) || ((2*x) < 0)。

假。设 x 等于-2147483648（$TMin_{32}$）。那么，我们将得到 2*x 等于 0。

B. (x & 7) != 7 || (x<<30 < 0).

真。如果(x & 7) != 7 这个表达式的值为 0，那么我们必须有位 x_2 等于 1。当左移 30 位时，这个位将变成符号位。

C. (x * x) >= 0。

假。当 x 为 65535（0xFFFF）时，x*x 为-131071（0xFFFE0001）。

D. x < 0 || -x <= 0。

真。如果 x 是非负数，则-x 是非正的。

E. x >> 0 || -x >= 0

假。设 x 为 2147483648（$TMin_{32}$）。那么 x 和-x 都为负数。

F. x*y == ux*uy。

真。二进制补码和无符号乘法有相同的位级行为。

G. ~x*y + uy*ux == -y。

真。~x 等于-x-1。uy*ux 等于 x*y。因此，左手边等价于-x*y-y+x*y。

练习题 2.31 答案

理解二进制小数表示是理解浮点编码的一个重要步骤。这个练习让你试验一些简单的例子。

小数值	二进制表示	十进制表示
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$	0.011	0.375
$\frac{23}{16}$	1.0111	1.4375
$\frac{77}{32}$	10.1101	2.40625

（续表）

小数值	二进制表示	十进制表示
$\frac{11}{8}$	1.011	1.375
$\frac{45}{8}$	101.101	5.625
$\frac{49}{16}$	11.0001	3.0625

考虑二进制小数表示的一个简单方法是将一个数表示为形如 $\frac{x}{2^k}$ 的小数。我们能够将这个形式表示为二进制，过程是：使用 x 的二进制表示，并把二进制小数点插入从右边算起的第 k 个位置。举一个例子，对于 $\frac{23}{16}$，我们有 $23_{10} = 10111_2$。然后我们把二进制小数点放在从右算起的第四位，得到 1.0111_2。

练习题 2.32 答案

在大多数情况中，浮点数的有限精度不是主要的问题，因为计算的相对错误仍然是相当低的。然而在这个例子中，系统对于绝对误差是很敏感的。

A. 我们可以看到$x-0.1$的二进制表示为：

$$0.00000000000000000000000001100[1100]\cdots_2$$

把这个表示与 $\frac{1}{10}$ 的二进制表示进行比较，我们可以看到这就是 $2^{-20}\times\frac{1}{10}$，也就是大约$9.54\times 10^{-8}$。

B. $9.54\times10^{-8}\times100\times60\times60\times10\approx0.343$。

C. $0.343\times2000\approx687$。

练习题 2.33 答案

研究非常小的字长的浮点表示能够帮助澄清IEEE浮点是怎样工作的。要特别注意非规格化数和规格化数之间的转换。

位	e	E	f	M	V
0 00 00	0	0	0	0	0
0 00 01	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
0 00 10	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$
0 00 11	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$

（续表）

位	e	E	f	M	V
0 01 00	1	0	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$
0 01 01	1	0	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
0 01 10	1	0	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$
0 01 11	1	0	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01	2	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$
0 10 10	2	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$
0 10 11	2	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

练习题 2.34 答案

十六进制 0x354321 等价于二进制[1101010100001100100001]。将之右移 21 位得到 $1.101010100001100100001_2 \times 2^{21}$。我们通过除去起始位的 1 并增加 2 个 0 形成小数域，从而得到[10101010000110010000100]。指数是通过 21 加上偏置量 127 形成的，得到 148（二进制[10010100]）。我们把它和符号域 0 联合起来，得到二进制表示

[01001010010101010000110010000100]

我们看到这两个表达式的相关性是，整数的低位到最高有效位等于 1，匹配小数的高 21 位：

```
    0   0   3   5   4   3   2   1
  00000000001101010100001100100001
            *********************
      4   A   5   5   0   C   8   4
     01001010010101010000110010000100
```

练习题 2.35 答案

这个练习帮助你思考什么数是不能用浮点准确表示的。

这个数的二进制表示是：1 后面跟着 n 个 0，其后再跟 1，得到值是 $2^{n+1}+1$。

当 $n=23$ 时，值是 $2^{24}+1=16\,777\,217$。

练习题 2.36 答案

一般来说，使用库宏（library macro）会比写你自己的代码更好一些。然而，这段代码似乎可以工作在多种机器上。

我们假设值 1e400 溢出为无穷。

code/data/ieee.c

```
1 #define POS_INFINITY 1e400
2 #define NEG_INFINITY (-POS_INFINITY)
3 #define NEG_ZERO (-1.0/POS_INFINITY)
```

code/data/ieee.c

练习题 2.37 答案

这样一个练习可以帮助你提高从程序员的角度来研究浮点运算的能力。

确信自己理解下面每一个答案。

A. `x == (int)(float) x`

错，例如当 x 为 *TMax* 时。

B. `x == (int)(double) x`

对，因为 double 类型比 int 类型具有更大的精度和范围。

C. `f == (float)(double) f`

对，因为 double 类型比 float 类型具有更大的精度和范围。

D. `d == (float) d`

错，例如，当 d 为 1e40 时，我们在右边得到 $+\infty$。

E. `f == -(-f)`

对，因为浮点数取非就是简单地对它的符号位取反。

F. `2/3 == 2/3.0`

错，左边的值将是整数值 0，而右边的值是浮点数 $\frac{2}{3}$ 的近似值。

G. `(d >= 0.0) || ((d*2) < 0.0)`

对，因为乘法是单调的。

H. `(d+f)-d == f`

错，例如当 d 是 $+\infty$ 而 f 是 1 时，左边将是 *NaN*，而右边将是 1。

CHAPTER 3

程序的机器级表示

在用高级语言如C语言编程时，我们被屏蔽了程序具体的机器级实现。相比之下，在用汇编代码写程序时，程序员必须明确指定程序该如何管理存储器（memory）和用来执行计算的低级指令。大多数时候，在高级语言提供的较高抽象级别上工作会更有成效和可靠。编译器提供的类型检查能帮助你发现许多程序错误，并能够保证我们是按照一致的方式来引用和管理数据的。使用现代的优化编译器，产生的代码通常至少与一个熟练的汇编语言程序员手工编写的代码一样有效。最好的一点就是，用高级语言编写的程序可以在很多不同的机器上编译执行，而汇编代码则是与特定机器密切相关的。

虽然可以使用优化编译器，但是对于严谨的程序员来说，能够阅读和理解汇编代码仍是一项很重要的技能。启动编译器时带上适当的选项，编译器就会产生一个汇编代码文件，汇编代码非常接近于计算机执行的实际机器代码。与目标代码的二进制格式相比，它的主要特色在于它采用的是更加易读的文本格式。通过阅读这些汇编代码，我们能够理解编译器的优化能力，并分析出代码中潜在的低效率。就像我们将在第5章中看到的那样，一个试图优化一段关键代码性能的程序员，通常会尝试源代码的各种形式，每次编译并检查产生出的汇编代码，从而了解程序将要运行的效率是如何的。此外，也有些时候，高级语言提供的抽象层会隐藏我们想要理解的一些信息，如程序的运行时行为。例如，第13章中会讲到，当用线程包写并发程序时，知道用何种存储（storage）来保存各种程序变量是很重要的，而这些信息在汇编代码级是可见的。程序员学习汇编代码的需求随着时间的推移也发生了变化，开始时是要求程序员能直接用汇编语言编写程序，现在则是要求他们能够阅读和理解优化编译器产生的代码。

在本章中，我们将学习某种汇编语言的详细内容，明白C程序是如何编译成这种形式的机器代码的。为了阅读编译器产生的汇编代码，除了具备手工编写汇编代码的能力外，还包括其他一些技能。我们必须了解典型的编译器在将C程序结构变换成机器代码时所做的转换。相对于C代码中表示的计算操作，优化编译器能够重新排列执行顺序，消除不必要的计算并替换慢速操作，例如用加法和移位来代替乘法，甚至于将递归计算变换成迭代计算。理解源代码与对应的汇编码的关系通常不太容易——就像要拼出一幅跟盒子上的图片设计有点不太一样的拼图。这是一种逆向工程（reverse engineering）——通过研究系统和逆向工作，来试着了解系统被创建的过程。在这个情况中，系统是一个机器产生的汇编语言程序，而不是由人设计的某个东西。这简化了逆向工程的任务，因为产生的代码遵循相当规则的模式，且我们可以做试验，让编译器产生许多不同程序的代码。在我们的表述中，给出了许多示例和练习，来说明汇编语言和编译器的各个方面。精通细节是理解更深和更基本概念的先决条件，花点时间研究这些示例并完成练习是非常值得的。

下面，我们简要回顾Intel的体系结构。Intel处理器从1978年那个相当简单的16位处理器发展而来，现在已经成为了桌面计算机的主流机器。随着新特性的加入，体系结构也在相应地成长，从16位体系结构转变成了支持32位数据和地址的结构。32位结构是相当奇怪的设计，有些特性只有从历史的角度来看才有意义。它还负担着提供后向兼容性的任务，这是现代编译器和操作系统不需要考虑的问题。我们将关注的是那些被GCC和Linux使用的特性的子集，这样可以避免许多复杂性以及IA32的隐秘特性。

我们的技术讲解是从快速浏览C、汇编代码以及目标代码之间的关系开始的。然后会讲到IA32的细节，从数据的表示和处理，及控制的实现开始。我们会看到如何实现C语言中的控制结构，如if、while和switch语句。这时，我们会讲到过程的实现，包括运行栈是如何支持过程间数据和控制

的传递，以及局部变量的存储（storage）。接着，我们会考虑在机器级如何实现像数组、结构和联合（union）这样的数据结构。有了这些机器级编程的背景知识，我们会看看存储器访问越界的问题，以及系统容易遭受缓冲区溢出攻击的问题。在这一部分的结尾，我们会给出一些用 GDB 调试器来检查机器级程序运行时行为的技巧。

接下来是标了星号“*”的内容，这是为专门的机器语言爱好者准备的。我们讲述了 IA32 对浮点代码的支持。这是 IA32 一个非常不可思议的特性，所以我们只建议那些决心要使用浮点代码的人来学习这个部分。我们还简要介绍了一下 GCC 对在 C 程序中嵌入汇编代码的支持。在某些应用程序中，程序员必须要用汇编代码来访问机器的某些低级特性。这时，嵌入汇编代码就是最好的方法。

3.1 历史观点

Intel 处理器系列的产生是一个长期的、不断进化的发展过程。它开始于一个单芯片、16 位微处理器，由于当时集成电路技术水平十分有限，其中不得不做了很多妥协。从此以后，它不断地成长，利用技术的进步去满足更高性能和支持更高级操作系统的需求。

下面的列表展示了按照时间顺序排列的 Intel 处理器模型，以及它们的一些关键特性。我们用实现这些处理器所需要的晶体管数量来表明它们复杂性的演变过程(K 表示 1000，而 M 表示 1000000)。

8086：（1978，29 K 个晶体管）。它是第一代单芯片、16 位微处理器之一。8088，即 8086 加上 8 位外部总线（external bus），构成最初的 IBM 个人计算机的心脏。IBM 与当时还很小的微软签订合同，开发 MS-DOS 操作系统。最初的机器型号有 32 768 字节的存储器和两个软驱（没有硬盘驱动器）。从体系结构上来说，这些机器只有 655 360 字节的地址空间——地址只有 20 位长（1 048 576 字节可被寻址），而操作系统保留了 393 216 字节自用。

80286：（1982，134K 个晶体管）。增加了更多的寻址模式（有些现在已经废弃了）。构成了 IBM PC-AT 个人计算机的基础，这种计算机是 MS Windows 最初的使用平台。

i386：（1985，275K 个晶体管）。将体系结构扩展到 32 位。增加了平面寻址模式（flat addressing model），Linux 和最近版本的 Windows 系列操作系统都是使用的这种模式。这是 Intel 系列中第一台支持 Unix 操作系统的机器。

i486：（1989，1.9M 个晶体管）。改善了性能，同时将浮点单元集成到处理器芯片上，但是没有改变指令集。

Pentium：（1993，3.1M 个晶体管）。改善了性能，不过只对指令集增加了小的扩展。

PentiumPro：（1995，6.5M 个晶体管）。引入全新的处理器设计，在内部被称为 P6 微体系结构。指令集中增加了一类“条件传送（conditional move）”指令。

Pentium/MMX：（1997，4.5M 个晶体管）。在 Pentium 处理器中增加了处理整数向量的新指令类。每个数据可以是 1、2 或 4 个字节长。每个向量总长 64 位。

Pentium II：（1997，7M 个晶体管）。通过在 P6 微体系结构中实现 MMX 指令，合并了以前分离的 PentiumPro 和 Pentium/MMX 系列。

Pentium III：（1999，8.2M 个晶体管）。引入另一类处理整数或浮点数向量的指令，每个数据可以是 1、2 或 4 个字节长，打包成 128 位的向量。由于在芯片上包括了二级高速缓存，这种芯片后来的版本最多使用了 24M 个晶体管。

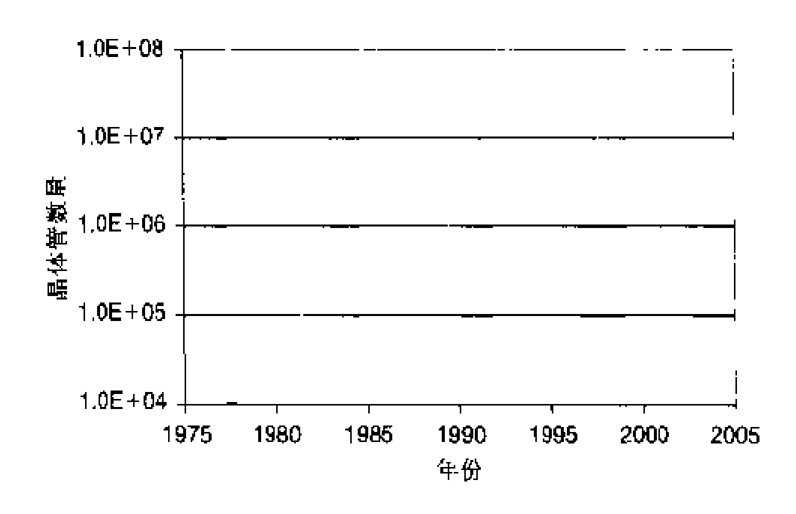
Pentium 4：（2001，42M 个晶体管）。在向量指令中增加了 8 字节整数和浮点格式，以及针对这些格式的 144 个新指令。在编号惯例上，Intel 不再使用罗马数字。

每个时间上相继的处理器设计都是后向兼容的——也就是，较早版本上编译的代码是可以在较新的处理器上运行的。正如我们会看到的那样，为了保持这种进化传统，指令集中有许多非常奇怪的东西。Intel 现在称其指令集为 IA32，也就是“Intel 32 位体系结构（Intel Architecture 32-bit）”。这个处理器系列也俗称为“x86”，反映出直到 i486 的处理器命名惯例。

旁注：为什么不叫 i586？

Intel 没有继续沿用他们的数字命名惯例，是因为他们无法获得 CPU 编号的商标保护。美国商标局不允许用数字作为商标。因此，他们创造了“Pentium”这个词，用的是希腊词根 penta，表明这是他们的第五代机器。从此以后，他们就使用这个词的变体，即使 PentiumPro 是第六代机器（因此内部称为 P6），而 Pentium 4 是第七代。每出现新的一代都包括处理器设计中的一个很大的变化。

旁注：摩尔定律（Moore's Law）



如果我们画出上面列出的各种 IA32 处理器中晶体管的数量与它们出现的年份之间的图，并且使 Y 轴为晶体管数的对数值，我们能够看出，增长是很显著的。划一条线穿过这些数据，我们看到晶体管数量以每年大约 33%的比率增加，也就是说，每 30 个月晶体管数量就会翻一番。在 IA32 的历史上，这种增长已经持续了大约 25 年。

1965 年，Gordon Moore，Intel 公司的创始人，根据当时芯片技术，也就是能够在一个芯片上制造有大约 64 个晶体管的电路，做出推断，预测在未来 10 年内，每年芯片上的晶体管数量都会翻一番。这个预测就称为摩尔定律。正如事实证明的那样，他的预测不仅有点乐观，而且太短视了。在它四十多年的历史里，半导体工业能够每 18 个月就将晶体管数目加倍。

对计算机技术的其他方面，也有类似的呈指数性增长的情况出现，比如磁盘容量，存储器芯片容量，和处理器性能。

这些年来，有几家公司生产出了与 Intel 处理器兼容的处理器，它们能够运行完全相同的机器级程序。其中，领头的是 AMD 公司。数年来，AMD 的策略一直是在技术上紧跟在 Intel 后面，生产性能稍低但是价格更便宜的处理器。最近，AMD 已经生产出了一些顶级性能的 IA32 处理器，这些

处理器是第一个突破商业可用微处理器 1G 时钟速度门槛的。虽然我们会谈到 Intel 处理器，但是这些描述对 Intel 的竞争对手生产的兼容处理器也同样适用。

对由 GCC 编译器产生出的、运行在 Linux 操作系统平台上的程序，感兴趣的人并没关注到 IA32 复杂性的大部分。最初的 8086 中的存储器模型和它在 80286 中的扩展都已经过时了。作为替代，Linux 使用了平面寻址方式（flat addressing），在这种寻址方式中，程序员将整个存储空间看做一个大的字节数组。

从列出的发展过程中，我们可以看到，IA32 中加入了很多处理小整数和浮点数向量的格式和指令。增加这些特性是为了提高多媒体应用程序的性能，例如图像处理、音频和视频编码和解码，以及三维计算机图形。不幸的是，目前版本的 GCC 产生的代码不会使用这些新特性。实际上，在默认启动方式下，GCC 会假设它是为一个 i386 机器产生代码，编译器不会试图使用许多添加到现在看来已经非常老的体系结构的扩展特性。

3.2 程序编码

假设我们写一个 C 程序，有两个文件 p1.c 和 p2.c。然后我们用 Unix 命令行编译这段代码：

```
unix> gcc -O2 -o p p1.c p2.c
```

命令 gcc 表明的就是 GNU C 编译器 GCC。因为这是 Linux 上默认的编译器，我们也可以简单地用 CC 来启动它。编译选项-O2 告诉编译器使用第二级优化。通常，提高优化级别会使最终程序运行得更快，但是编译时间可能会变长，对代码进行调试会更困难。第二级优化是性能优化和使用方便之间的一种很好的妥协。本书中所有的代码都是用这个优化级别进行编译的。

这个命令实际上调用了一系列程序，将源代码转化成可执行代码，首先，C 预处理器会扩展源代码，插入所有用#include 命令指定的文件，并扩展所有的宏。其次，编译器产生两个源文件的汇编代码，名字分别为 p1.s 和 p2.s。接下来，汇编器会将汇编代码转化成二进制目标代码文件 p1.o 和 p2.o。最后，链接器将两个目标文件与实现标准 Unix 库函数（例如 printf）的代码合并，并产生最终的可执行文件。我们会在第 7 章中更详细地介绍链接。

3.2.1 机器级代码

在整个编译过程中，编译器会完成大部分的工作，将把用 C 提供的相对比较抽象的执行模型表示的程序转化成处理器执行的非常基本的指令。汇编代码表示非常接近于机器代码。与目标代码的二进制格式相比，汇编代码的主要特点是用可读性更好的文本格式表示的。能够理解汇编代码以及它是如何与原始的 C 代码相对应的，是理解计算机如何执行程序的关键一步。

汇编程序员看到的机器与 C 程序员看到的机器差别很大。一些通常对 C 程序员屏蔽的处理器状态是可见的：

- 程序计数器（称为%eip）表示将要执行的下一条指令在存储器中的地址。
- 整数寄存器文件包含 8 个被命名的位置，分别存储 32 位的值。这些寄存器可以存储地址（对应于 C 的指针）或整数数据。有的寄存器用来记录某些重要的程序状态，而其他的寄存器用来保存临时数据，例如过程的局部变量。
- 条件码寄存器保存着最近执行的算术指令的状态信息。它们用来实现控制流中的条件变化，

比如说用来实现 if 或 while 语句。

- 浮点寄存器文件包含 8 个位置，用来存放浮点数据。

虽然 C 提供了一种模型，可以在存储器中声明和分配各种数据类型的对象，但是汇编代码只是简单地将存储器看成一个很大的、按字节寻址的数组。C 中的聚集数据类型，例如数组和结构，在汇编代码中是用连续的字节表示的。即使是对标量数据类型，汇编代码也不区分有符号或无符号整数，不区分各种类型的指针，甚至于不区分指针和整数。

程序存储器（program memory）包含程序的目标代码，操作系统需要的一些信息，用来管理过程调用和返回的运行时栈，以及用户分配的存储器块（比如说用 malloc 库函数分配的）。

程序存储器是用虚拟地址来寻址的。在任意给定的时刻，只有有限的一部分虚拟地址是合法的。例如，虽然 IA32 的 32 位地址可以寻址 4GB 的地址范围，但是一个通常的程序只会访问几 M 字节。操作系统负责管理虚拟地址空间，将虚拟地址转换成实际处理器存储器（processor memory）中的物理地址。

一条机器指令只执行非常基本的操作。例如，将两个存放在寄存器中的数字相加，在存储器和寄存器之间传递数据，或是条件分支转移到新的指令地址。编译器必须产生这些指令序列，从而实现象算术表达式求值、循环或过程调用和返回这样的程序结构。

3.2.2 代码示例

假设我们写了一个 C 代码文件 code.c，包含下面这样的过程定义：

```
1   int accum = 0;
2
3   int sum(int x, int y)
4   {
5       int t = x + y;
6       accum += t;
7       return t;
8   }
```

在命令行上使用“-S”选项，就能看到 C 编译器产生的汇编代码：

```
unix> gcc -O2 -S code.c
```

这会使编译器产生一个汇编文件 code.s，但是不做其他进一步的工作（通常情况下，它还会调用汇编器产生目标代码文件）。

GCC 是按照它自己的格式产生汇编代码的，这种格式称为 GAS（Gnu ASsembler，GNU 汇编器）。我们的讲述是基于这种格式的，它同 Intel 文档中的格式以及微软编译器使用的格式差异很大。从参考文献说明中可以获得关于如何找到各种汇编代码格式文档的建议。

汇编代码文件包含各种声明，包括下面所示：

```
sum:
  pushl %ebp
  movl %esp,%ebp
  movl 12(%ebp),%eax
  addl 8(%ebp),%eax
```

```
 addl %eax,accum
 movl %ebp,%esp
 popl %ebp
 ret
```

上面代码中每个缩进去的行都对应于一条机器指令。比如，pushl 指令表示应该将寄存器%ebp的内容压入程序栈中。这段代码中已经除去了所有关于局部变量名或数据类型的信息，但我们还是看到了一个对全局变量 accum 的引用，这是因为编译器还不能确定这个变量会放在存储器中的哪个位置。

如果我们使用“-c”命令行选项，GCC 会编译并汇编该代码：

```
unix> gcc -O2 -c code.c
```

这就会产生目标代码文件 code.o，它是二进制格式的，所以无法直接读。852 字节的文件 code.o 中有一段 19 字节的十六进制表示的序列：

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 89 ec 5d c3
```

这就是对应于上面列出的汇编指令的目标代码。从中得到的重要信息就是，机器实际执行的程序只是对一系列指令进行编码的字节序列。机器对产生这些指令的源代码几乎一无所知。

旁注：如何找到程序的字节表示？

首先，我们用反汇编器（待会儿会讲到的）来确定函数 sum 的代码长是 19 字节。然后，我们在文件 code.o 上运行 GNU 调试工具 GDB，输入命令：

```
(gdb) x/19xb sum
```

这条命令告诉 GDB 检查（简写为“x”）19 个十六进制格式（也简写为“x”）的字节（简写为“b”）。你会发现，GDB 有很多有用的特性可以用来分析机器级程序，我们会在 3.12 节中讨论这个问题。

要查看目标代码文件的内容，有一类称为反汇编器（disassembler）的程序的价值无法估量，这些程序根据目标代码生成一种类似于汇编代码的格式。在 Linux 系统中，带“-d”命令行选项的程序 OBJDUMP（代表“object dump”）可以充当这个角色：

```
unix> objdump -d code.o
```

结果是（这里，我们在左边增加了行号，在右边增加了注解）：

```
     Disassembly of function sum in file code.o
1    00000000 <sum>:
       Offset Bytes                    Equivalent assembly language
2        0:  55                          push    %ebp
3        1:  89 e5                       mov     %esp,%ebp
4        3:  8b 45 0c                    mov     0xc(%ebp),%eax
5        6:  03 45 08                    add     0x8(%ebp),%eax
6        9:  01 05 00 00 00 00           add     %eax,0x0
7        f:  89 ec                       mov     %ebp,%esp
8       11:  5d                          pop     %ebp
9       12:  c3                          ret
10      13:  90                          nop
```

在左边，我们看到按照前面给出的字节顺序排列的19个十六进制字节值，它们分成了一些组，每组有1~6个字节。每组都是一条指令，右边是等价的汇编语言。其中一些特性值得说明：

- IA32指令长度从1~15个字节不等。指令编码被设计成使常用的指令以及操作数较少的指令所需的字节数少，而那些不太常用或操作数较多的指令所需字节数较多。
- 指令格式是按照这样一种方式设计的，从某个给定位置开始，可以将字节惟一地解码成机器指令。例如，只有指令pushl %ebp是以字节值55开头的。
- 反汇编器只是根据目标文件中的字节序列来确定汇编代码的。它不需要访问程序的源代码或汇编代码。
- 反汇编器使用的指令命名规则与GAS使用的有些细微的差别。在我们的示例中，它省略了很多指令结尾的“l”。
- 与code.s中的汇编代码相比，我们还发现结尾多了一条nop指令。这条指令根本不会被执行（它在过程返回指令之后），即使执行了也不会有任何影响（所以称之为nop，是“no operation”的简写，通常读作“no op”）。编译器插入这样的指令是为了填充存储该过程的空间。

生成实际可执行的代码需要对一组目标代码文件运行链接器，而这一组目标代码文件中必须含有一个main函数。假设在文件main.c中有下面这样的函数：

```
1    int main()
2    {
3        return sum(1, 3);
4    }
```

然后，我们用如下方法生成可执行文件test：

```
unix> gcc -O2 -o prog code.o main.c
```

文件prog变成了11667字节，因为它不仅包含我们的两个过程的代码，还包含了用来启动和终止程序的信息，以及用来与操作系统交互的信息。我们也可以反汇编prog文件：

```
unix> objdump -d prog
```

反汇编器会抽取出各种代码序列，包括下面这段：

Disassembly of function sum *in executable file* prog

```
1    080483b4 <sum>:
2     80483b4:   55                     push   %ebp
3     80483b5:   89 e5                  mov    %esp,%ebp
4     80483b7:   8b 45 0c               mov    0xc(%ebp),%eax
5     80483ba:   03 45 08               add    0x8(%ebp),%eax
6     80483bd:   01 05 64 94 04 08      add    %eax,0x8049464
7     80483c3:   89 ec                  mov    %ebp,%esp
8     80483c5:   5d                     pop    %ebp
9     80483c6:   c3                     ret
10    80483c7:   90                     nop
```

注意，这段代码与code.c反汇编产生的代码几乎完全一样。一个主要的区别是左边列出的地址不同——链接器将代码的地址移到一段不同的地址范围。第二个不同之处在于链接器终于确定存储全局变量accum的地址。code.o反汇编代码的第6行中，accum的地址还是0。prog的反汇编代码

中，地址就设成了 0x8049464。这可以从指令的汇编代码格式中看到，还可以从指令的最后四个字节中看出来，从最低位到最高位列出的就是 64 94 04 08。

3.2.3 关于格式的注解

GCC 产生的汇编代码有点难读，它包含一些我们不需要关心的信息。另外，它不提供任何程序的描述或它是如何工作的描述。例如，假设文件 simple.c 包含下列代码：

```
1    int simple(int *xp, int y)
2    {
3       int t = *xp + y;
4       *xp = t;
5       return t;
6    }
```

当带选项"-S"运行 GCC 时，它产生下面的文件 simple.s：

```
  .file  "simple.c"
  .version      "01.01"
gcc2_compiled.:
.text
  .align 4
.globl simple
  .type    simple,@function
simple:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl (%eax),%edx
  addl 12(%ebp),%edx
  movl %edx,(%eax)
  movl %edx,%eax
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size    simple,.Lfe1-simple
  .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

文件包含的信息多于我们实际需要的。所有以"."开头的行都是指导汇编器和链接器的命令（directive），不过我们通常可以忽略这些行。另一方面，也没有关于这些指令是干什么用的以及它们与源代码之间关系的解释说明。

为了更清楚地说明汇编代码，我们将给出汇编代码的格式，包括行号和解释性说明。对于我们的示例，带解释的汇编代码是像下面这样的：

```
1    simple:
2      pushl %ebp              Save frame pointer
3      movl %esp,%ebp          Create new frame pointer
4      movl 8(%ebp),%eax       Get xp
5      movl (%eax),%edx        Retrieve *xp
```

```
6      addl 12(%ebp),%edx     Add y to get t
7      movl %edx,(%eax)       Store t at *xp
8      movl %edx,%eax         Set t as return value
9      movl %ebp,%esp         Reset stack pointer
10     popl %ebp              Reset frame pointer
11     ret                    Return
```

通常我们只会给出与要讨论内容相关的代码行。每一行的左边都有编号供引用，右边是注释，简单地描述指令的效果以及它与原始C代码中的计算操作的关系。这是一种汇编语言程序员写代码的风格。

3.3 数据格式

由于是从16位体系结构扩展成32位的，Intel用术语“字（word）”表示16位数据类型。因此，称32位数为“双字（double words）”，称64位数为“四字（quad words）”。我们将遇到的大多数指令都是对字节或双字操作的。

图3.1给出了对应C基本数据类型的机器表示。注意，大多数常用数据类型都是作为双字存储的。其中，包括普通整数（int）和长整数（long int），无论它们是否有符号。此外，所有的指针（在此用char *表示）都是4字节的双字。处理字符串数据时，通常用到字节。浮点数有三种形式：单精度（4字节）值，对应于C数据类型float；双精度（8字节）值，对应于C数据类型double；和扩展精度（10字节）值。GCC用数据类型long double来表示扩展精度的浮点值。为了提高存储器系统的性能，它将这样的浮点数存储成12字节数，待会儿我们会讨论这个问题。虽然ANSI C标准包括long double数据类型，但是对大多数编译器和机器组合来说，它的实现和普通double的8字节格式是一样的。对GCC和IA32的组合来说，支持扩展精度是很少见的。

C声明	Intel数据类型	GAS后缀	大小（字节）
char	字节	b	1
short	字	w	2
int	双字	l	4
unsigned	双字	l	4
long int	双字	l	4
unsigned long	双字	l	4
char *	双字	l	4
float	单精度	s	4
double	双精度	l	8
long double	扩展精度	t	10/12

图3.1 标准数据类型的大小

如图3.1所示，GAS中的每个操作都有一个字符后缀，表明操作数的大小。例如，mov（传送数据）指令有三种形式：movb（传送字节）、movw（传送字）和movl（传送双字）。后缀“l”用来表示双字，因为在许多机器上，32位数都称为“长字（long word）”，这是沿用以16位字为标准的

时代的习惯造成的。注意，GAS 使用后缀“l”来同时表示 4 字节的整数和 8 字节的双精度浮点数。这不会产生歧义，因为浮点数使用的是一组完全不同的指令和寄存器。

3.4 访问信息

一个 IA32 中央处理单元（CPU）包含一组八个存储 32 位值的寄存器，这些寄存器用来存储整数数据和指针。图 3.2 显示了这八个寄存器。它们的名字都是以%e 开头的，不过它们都有特殊的名字。在最初的 8086 中，寄存器是 16 位的，每个都有特殊的用途。选择的名字就是用来反映各种用途的。在平面寻址中，对特殊寄存器的需求已经大为降低了。在大多数情况中，前六个寄存器都可以看成通用寄存器，对它们的使用没有限制。我们说“在大多数情况中”，是因为有些指令是以固定的寄存器作为源和/或目的的。另外，在过程（procedures）处理中，对前三个寄存器（%eax、%ecx 和%edx）的保存和恢复惯例将不同于接下来的三个寄存器（%ebx、%edi 和%esi），我们会在 3.7 节中对此加以讨论。最后两个寄存器（%ebp 和%esp）保存着指向程序栈中重要位置的指针，只有根据栈管理的标准惯例才能修改这两个寄存器中的值。

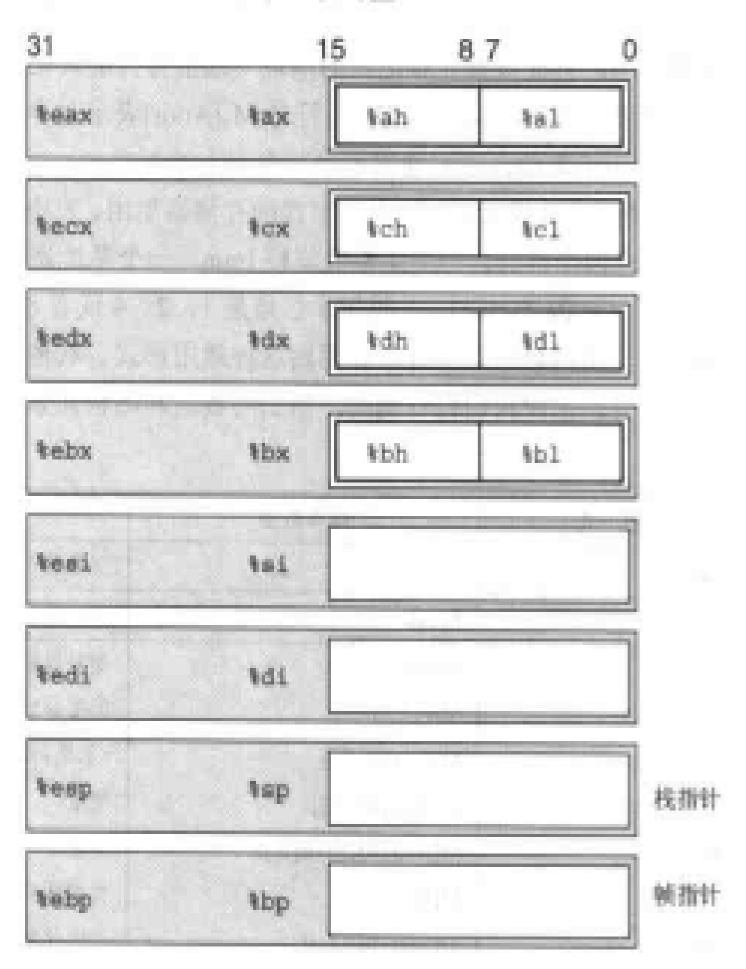


图 3.2 整数寄存器

所有八个寄存器都可以作为 16 位（字）或 32 位（双字）来访问，也可以独立访问前四个寄存器的两个低位字节。

如图 3.2 所示，字节操作指令可以独立地读或者写前四个寄存器的两个低位字节。8086 中提供

这样的特性是为了后向兼容 8008 和 8080，8008 和 8080 是两款可以追述到 1974 年的微处理器。当一条字节指令更新这些单字节"寄存器元素"中的一个时，该寄存器余下的三个字节不会被改变。类似地，字操作指令可以读或者写每个寄存器的低 16 位。这个特性源自 IA32 是从 16 位微处理器演化而来的。

3.4.1 操作数指示符

大多数指令有一个或多个操作数（operand），指示出执行一个操作中要引用的源数据值，以及放置结果的目的位置。IA32 支持多种操作数格式（图 3.3）。源数据值可以以常数形式给出，或是从寄存器或存储器中读出，结果可以存放在寄存器或存储器中。因此，各种操作数的可能性被分为三种类型。第一种是立即数（immediate），也就是常数值。在 GAS 中，采用标准 C 的表示方法，立即数的书写方式是"$"后面跟一个整数，比如，$-577 或$0x1F。任何 32 位的字都可以用做立即数，不过汇编器在可能时会使用一个或两个字节的编码。第二种类型是寄存器（register），它表示某个寄存器的内容，对双字操作来说，可以是八个 32 位寄存器中的一个（如%eax），对字节操作来说，可以是八个单字节寄存器元素中的一个（如%al）。在我们的图中，我们用符号 E_a 来表示任意寄存器 a，用引用 $R[E_a]$ 来表示它的值，这是将寄存器集合看成一个数组 R，用寄存器标识符作为索引。

第三类操作数是存储器引用，它会根据计算出来的地址（通常称为有效地址）访问某个存储器位置。因为将存储器看成一个很大的字节数组，我们用符号 $M_b[Addr]$ 表示对存储在存储器中从地址 Addr 开始的 b 字节值的引用。为了简便，我们通常省去写在下方的 b。

如图 3.3 所示，有多种不同的寻址模式，允许不同形式的存储器引用。表中底部的 $Imm(E_b, E_i, s)$ 是最通常的形式。这样的引用有四个部分：一个立即数偏移 Imm，一个基址寄存器 E_b，一个变址或索引寄存器 E_i 和一个伸缩因子（scale factor）s，这里 s 必须是 1、2、4 或者 8。然后，有效地址被计算为 $Imm + R[E_b] + R[E_i] \cdot s$。引用数组元素时，会用到这种通用形式。其他形式只是这种通用形式的特殊情况，省略了某些部分。正如我们将看到的，当引用数组和结构元素时，比较复杂的寻址模式是很有用的。

类 型	格 式	操作数值	名 称
立即数	$Imm	Imm	立即数寻址
寄存器	E_a	$R[E_a]$	寄存器寻址
寄存器	Imm	$M[Imm]$	绝对寻址
寄存器	(E_a)	$M[R[E_a]]$	间接寻址
寄存器	$Imm(E_b)$	$M[Imm+R[E_b]]$	（基址+偏移量）寻址
寄存器	(E_b, E_i)	$M[R[E_b]+R[E_i]]$	变址
寄存器	$Imm(E_b, E_i)$	$M[Imm+R[E_b]+R[E_i]]$	寻址
寄存器	$(, E_i, s)$	$M[R[E_i] \cdot s]$	伸缩化的变址寻址
寄存器	$Imm(, E_i, s)$	$M[Imm+R[E_i] \cdot s]$	伸缩化的变址寻址
寄存器	(E_b, E_i, s)	$M[R[E_b]+R[E_i] \cdot s]$	伸缩化的变址寻址
寄存器	$Imm(E_b, E_i, s)$	$M[Imm+R[E_b]+R[E_i] \cdot s]$	伸缩化的变址寻址

图 3.3 操作数格式

操作数可以表示立即数（常数）值、寄存器值或是来自存储器的值。伸缩因子 s 必须是 1、2、4 或者 8。

练习题 3.1

假设下面的值存放在指明的存储器地址和寄存器中：

地址	值
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

寄存器	值
%eax	0x100
%ecx	0x1
%edx	1x3

填写下表，给出所示操作数的值：

操作数	值
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFc(,%ecx,4)	
(%eax,%edx,4)	

3.4.2 数据传送指令

最频繁使用的指令是执行数据传送的指令。操作数符号的通用性使得一条简单的传送指令能够完成许多机器中要好几条指令才能完成的功能。图 3.4 列出的是一些重要的数据传送指令，最常用的是传送双字的 movl 指令。源操作数指定一个值，它可以是立即数，可以存放在寄存器中，也可以存放在存储器中。目的操作数指定一个位置，它可以是寄存器，也可以是存储器地址。IA32 加了一条限制，传送指令的两个操作数不能都指向存储器位置。将一个值从一个存储器位置拷到另一个存储器位置需要两条指令——第一条指令将源值加载到寄存器中，第二条将该寄存器值写入目的位置。

下面这个 movl 指令示例给出了源和目的类型的五种可能组合。回想一下，第一个是源操作数，第二个是目的操作数：

```
1      movl $0x4050,%eax          Immediate--Register
2      movl %ebp,%esp             Register--Register
3      movl (%edi,%ecx),%eax      Memory--Register
4      movl $-17,(%esp)           Immediate--Memory
5      movl %eax,-12(%ebp)        Register--Memory
```

movb 指令是类似的，除了它只传送一个字节。当一个操作数是寄存器时，它必须是图 3.2 中所示的八个单字节寄存器元素中的一个。类似地，movw 指令传送两个字节，当它的一个操作数为寄存器时，它必须是图 3.2 中所示的八个两字节寄存器元素中的一个。

指令		效果	描述
movl	S, D	D ← S	传送双字
movw	S, D	D ← S	传送字
movb	S, D	D ← S	传送字节
movsbl	S, D	D ← 符号扩展 (S)	传送符号扩展的字节
movzbl	S	D ← 零扩展 (S)	传送零扩展的字节
pushl	S	R[%esp] ← R[%esp]-4; M[R[%esp]] ← S	压栈
popl	D	D ← M[R[%esp]]; R[%esp] ← R[%esp]+4	出栈

图 3.4 数据传送指令

movsbl 和 movzbl 指令负责拷贝一个字节，并设置目的操作数中其余的位。movsbl 指令的源操作数是单字节的，它执行符号扩展到 32 位（也就是，将高 24 位设置为源字节的最高位），然后拷贝到双字的目的中。类似地，movzbl 指令的源操作数是单字节的，在前面加 24 个 0 扩展到 32 位，并将结果拷贝到双字的目的中。

旁注：字节传送指令比较

仔细观察可以发现，三个字节传送指令 movb、movsbl 和 movzbl 之间有细微的差别。这里有一个示例：

```
    初始假设 %dh = 8D, %eax = 98765432
1   movb %dh,%al            %eax = 9876548D
2   movsbl %dh, %eax        %eax = FFFFFF8D
3   movzbl %dh,%eax         %eax = 0000008D
```

在这些例子中，都是将寄存器%eax 的低位字节设置成%edx 的第二个字节。movb 指令不改变其他三个字节。根据源字节的最高位，movsbl 指令将其他三个字节设为全 1 或全 0。movzbl 指令无论如何都是将其他三个字节设为全 0。

最后两个数据传送操作是用来将数据压入栈中和从栈中弹出数据的。正如我们将看到的，栈在处理过程调用中起到至关重要的作用。pushl 和 popl 指令都只有一个操作数——用于压入的源数据和用于弹出的目的数据。程序栈存放在存储器中某个区域。如图 3.5 所示，栈向下增长，这样一来，栈顶元素的地址是所有栈中元素地址中最低的。（根据惯例，我们的栈是倒过来画的，栈“顶”在图的底部。）栈指针%esp 保存着栈顶元素的地址。将一个双字值压入栈中，首先要将栈指针减 4，然后将值写到新的栈顶地址。因此，指令 pushl %ebp 的行为等价于下面这样两条指令：

```
subl $4,%esp
movl %ebp,(%esp)
```

它们之间的区别是在目标代码中 pushl 指令是编码为 1 个字节的，而上面那两条指令一共需要 6 个字节。图中前两栏给出的是当%esp 为 0x108 和%eax 为 0x123 时，执行指令 pushl %eax 的效果。首先%esp 会减 4，得到 0x104，然后会将 0x123 存放到存储器地址 0x104 处。

弹出一个双字这样的操作将包括从栈顶位置读出数据，然后将栈指针加 4。因此，指令 popl %eax

等价于下面这样两条指令：

```
movl (%esp),%eax
addl $4,%esp
```

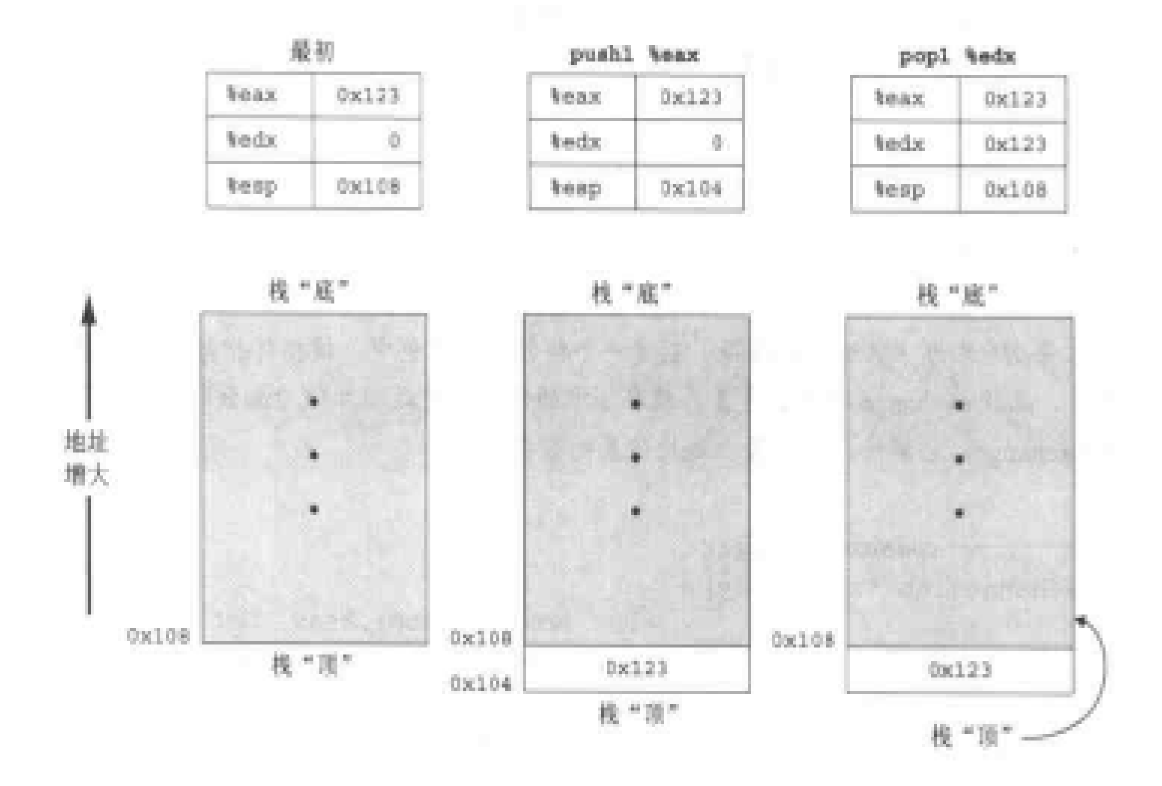


图 3.5 栈操作说明

根据惯例，我们的栈是倒过来画的，因而栈"顶"在底部。IA32 的栈向低地址方向增长，所以压栈是减小栈指针（寄存器%esp）的值，并存放到存储器中，而出栈是从存储器中读，并增加栈指针的值。

图 3.5 的第三栏说明的是在执行完 pushl 后立即执行指令 popl %edx 的效果。先从存储器中读出值 0x123，再写到寄存器%edx 中，然后，寄存器%esp 的值将增加为 0x108。如图中所示，值 0x123 仍然会保持在存储器位置 0x104 中，直到被另一条入栈操作覆盖。无论如何，%esp 指向的地址总是栈顶。

因为栈和程序代码以及其他形式的程序数据都是放在同样的存储器中，所以程序可以用标准的存储器寻址方法访问栈内任意位置。例如，假设栈顶元素是双字，指令 `movl 4(%esp), %edx` 会将第二个双字从栈中拷贝到寄存器%edx。

3.4.3 数据传送示例

给 C 语言初学者：一些指针的示例

函数 exchange（图 3.6）提供了一个关于 C 中指针使用的很好说明。参数 xp 是一个指向整数的指针，而 y 是一个整数。语句

```
int x = *xp;
```

表示我们将读存储在 xp 所指位置中的值，并将它存放到名字为 x 的局部变量中。这个读操作称为指针的间接引用（pointer dereferencing），C 操作符*执行指针的间接引用。

语句

```
*xp = y;
```

正好相反——它将参数 y 的值写到 xp 所指的位置。这也是一种指针间接引用的形式（所以有操作符*），但是它表明的是一个写操作，因为它是在赋值语句的左边。

下面是一个使用 exchange 的例子：

```
int a = 4;
int b = exchange(&a, 3);
printf("a = %d, b = %d\n", a, b);
```

这段代码会打印出：

```
a = 3, b = 4
```

C 操作符&（称为"取址"操作符）创建一个指针，在本例中，该指针指向保存局部变量 a 的位置。然后，函数 exchange 将用 3 覆盖存储在 a 中的值，但是返回 4 作为函数的值。注意如何将指针传递给 exchange，它能修改存在某个远处位置的数据。

code/asm/exchange.c

```
1  int exchange(int *xp, int y)
2  {
3      int x = *xp;
4
5      *xp = y;
6      return x;
7  }
```

code/asm/exchange.c

(a) C 代码

```
1   movl 8(%ebp),%eax    Get xp
2   movl 12(%ebp),%edx   Get y
3   movl (%eax),%ecx     Get x at *xp
4   movl %edx,(%eax)     Store y at *xp
5   movl %ecx,%eax       Set x as return value
```

(b) 汇编代码

图 3.6 exchange 函数体的 C 和汇编代码

省略了栈的建立和完成部分。

作为一个使用数据传送指令的代码示例，考虑图 3.6 中所示的数据交换函数，既有 C 代码，也有 GCC 产生的汇编代码。我们省略了过程入口处的汇编代码，这些代码用来为运行时栈分配空间，以及在过程返回前回收栈空间的代码。当我们讨论过程链接时，会讲到这种建立和完成代码的细节。除此之外剩下的代码，我们称之为"过程体（body）"。

当过程体开始执行时，过程参数 xp 和 y 存储在相对于寄存器%ebp 中地址值的偏移 8 和 12 的地方。指令 1 和 2 会将这些参数传送寄存器%eax 和%edx。指令 3 间接引用 xp，并将值存储在寄存器%ecx 中，对应于程序值 x。指令 4 将 y 存储在 xp。指令 5 将 x 传送到寄存器%eax。根据惯例，所有返回整数或指针值的函数都是通过将结果放在寄存器%eax 中来达到目的的，因此这条指令实现了 C 代码中第 6 行的功能。这个例子说明 movl 指令是如何用于从存储器中读值到寄存器的（指令 1～3），如何从寄存器写到存储器的（指令 4），以及如何从一个寄存器拷贝到另一个寄存器的（指令 5）。

关于这段汇编代码有两点值得注意。首先，我们看到 C 中所谓的"指针"其实就是地址。间

接引用指针就是将该指针放在一个寄存器中，然后在间接存储器引用中使用这个寄存器。其次，像 x 这样的局部变量通常是保存在寄存器中，而不是存储器中。寄存器访问比存储器访问要快得多。

练习题 3.2

题设信息如下，将一个原型为

```
void decode1 (int *xp, int *yp, int *zp);
```

的函数编译成汇编代码。代码体如下：

```
1       movl 8(%ebp),%edi
2       movl 12(%ebp),%ebx
3       movl 16(%ebp),%esi
4       movl (%edi),%eax
5       movl (%ebx),%edx
6       movl (%esi),%ecx
7       movl %eax,(%ebx)
8       movl %edx,(%esi)
9       movl %ecx,(%edi)
```

参数 xp、yp 和 zp 存储在相对于寄存器%ebp 中地址值的偏移 8、12 和 16 的地方。

请写出等效于上面汇编代码的 decode1 的 C 代码。可以用-S 选项编译你的代码，检验你的答案。你的编译器生成的代码在寄存器的使用或是存储器引用的顺序上可能会有所不同，但是功能应该是等效的。

3.5 算术和逻辑操作

图 3.7 列出了一些双字整数操作，分为四类。二元操作有两个操作数，而一元操作只有一个操作数。描述这些操作数的符号与 3.4 节中使用的符号完全相同。除了 leal 以外，每条指令都有对应的对字（16 位）和对字节操作的指令。把后缀 “l” 换成 “w” 就是对字的操作，而换成 “b” 就是对字节的操作了。例如，addl 对应有 addw 和 addb。

3.5.1 加载有效地址

加载有效地址（Load Effective Address）指令 leal 实际上是 movl 指令的变形。它的指令形式是从存储器读数据到寄存器，但实际上它根本就没引用存储器。它的第一个操作数看上去是一个存储器引用，但该指令并不是从指定的位置读入数据，而是将有效地址写入到目的操作数（如寄存器）。在图 3.7 中我们用 C 的地址操作符&S 来说明这种计算。这条指令可以用来为后面的存储器引用产生指针。另外，它还可以用来简洁地描述普通的算术操作。例如，如果寄存器%edx 的值为 x，那么指令 `leal 7(%edx, %edx, 4), %eax` 将设置寄存器%eax 的值为 5x + 7。注意：目的操作数必须是一个寄存器。

练习题 3.3

假设寄存器%eax 的值为 x，%ecx 的值为 y。填写下表，指明下面每条汇编代码指令存储在寄存器%edx 中的值。

表达式	结果
`leal 6(%eax),%edx`	
`leal (%eax,%ecx),%edx`	
`leal (%eax,%ecx,4),%edx`	
`leal 7(%eax,%ecx,8),%edx`	
`leal 0xA(,%eax,4),%edx`	
`leal 9(%eax,%ecx,2),%edx`	

指令		效果	描述
leal	S, D	$D \leftarrow \&S$	加载有效地址
incl	D	$D \leftarrow D+1$	加1
decl	D	$D \leftarrow D-1$	减1
negl	D	$D \leftarrow -D$	取负
notl	D	$D \leftarrow \sim D$	取补
addl	S, D	$D \leftarrow D+S$	加
subl	S, D	$D \leftarrow D-S$	减
imull	S, D	$D \leftarrow D*S$	乘
xorl	S, D	$D \leftarrow D\wedge S$	异或
orl	S, D	$D \leftarrow D\mid S$	或
andl	S, D	$D \leftarrow D\&S$	与
sall	k, D	$D \leftarrow D<<k$	左移
shll	k, D	$D \leftarrow D<<k$	左移（等同于 sall）
sarl	k, D	$D \leftarrow D>>k$	算术右移
shrl	k, D	$D \leftarrow D>>k$	逻辑右移

图 3.7 整数算术操作

加载有效地址（leal）指令通常用来执行简单的算术操作，而其余的指令是非常标准的一元或二元操作。注意，GAS 中的操作数顺序与上表相反。

3.5.2 一元和二元操作

第二类操作是一元操作，只有一个操作数，既作源，也作目的。这个操作数可以是一个寄存器，也可以是一个存储器位置。比如说，指令 `incl(%esp)` 会使栈顶元素加 1。这种语法让人想起 C 中的加 1 运算符（++）和减 1 运算法（--）。

第三类是二元操作，第二个操作数既是源又是目的。这种语法让人想起 C 中像+=这样的赋值运算符。不过，要注意，源操作数是第一个，目的操作数是第二个，这是不可交换操作特有的。例如，指令 `subl %eax, %edx` 使寄存器%edx 的值减去%eax 中的值。第一个操作数可以是立即数、寄存器或是存储器位置。第二个操作数可以是寄存器或是存储器位置。不过，同 movl 指令一样，两个操作数不能同时都是存储器位置。

练习题 3.4

假设下面的值存放在指定的存储器地址和寄存器中：

地址	值
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

寄存器	值
%eax	0x100
%ecx	0x1
%edx	0x3

填写下表，给出下面指令的效果，要说明将被更新的寄存器或存储器位置，以及得到的值：

指　令	目　的	值
addl %ecx,(%eax)		
subl %edx,4(%eax)		
imull $16,(%eax,%edx,4)		
incl 8(%eax)		
decl %ecx		
subl %edx,%eax		

3.5.3 移位操作

最后一类是移位操作，先给出移位量，然后是待移位的值。可以进行算术和逻辑右移。移位量用单个字节编码，因为只允许进行 0 到 31 位的移位。移位量可以是一个立即数，或者放在单字节寄存器元素%cl 中。如图 3.7 所示，左移指令有两个名字：sall 和 shll。两者的效果都一样，都是将右边填上 0。右移指令不同，sarl 执行算术移位（填上符号位），而 shrl 执行逻辑移位（填上 0）。

练习题 3.5

假设我们想生成下面这个 C 函数的汇编代码：

```
int shift_left2_rightn(int x, int n)
{
  x <<= 2;
  x >>= n;
  return x;
}
```

下面这段代码执行实际的移位，并将最后的结果放在寄存器%eax 中。此处省略了两条重要的指令。参数 x 和 n 分别存放在存储器中相对于寄存器%ebp 中地址偏移 8 和 12 的地方。

```
1    movl 12(%ebp),%ecx     Get n
2    movl 8(%ebp),%eax      Get x
3    ______________         x <<= 2
4    ______________         x >>= n
```

根据右边的注释，填出缺失的指令。请用算术右移操作。

3.5.4 讨论

除了右移操作，所有的指令都不区分有符号和无符号操作数。对列出的所有指令来说，二进制补码运算和无符号运算有同样的位级行为。

图 3.8 给出了一个执行算术操作的函数示例，以及它的汇编代码。和前面一样，我们省略了栈的建立和完成部分。函数参数 x、y 和 z 分别存放在存储器中相对于寄存器%ebp 中地址偏移 8、12 和 16 的地方。

code/asm/arith.c

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x-y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9
10      return t4;
11  }
```

code/asm/arith.c

(a) C 代码

```
1   movl 12(%ebp),%eax          Get y
2   movl 16(%ebp),%edx          Get z
3   addl 8(%ebp),%eax           Compute t1 = x+y
4   leal (%edx,%edx,2),%edx     Compute z*3
5   sall $4,%edx                Compute t2 = z*48
6   andl $65535,%eax            Compute t3 = t1&0xFFFF
7   imull %eax,%edx             Compute t4 = t2*t3
8   movl %edx,%eax              Set t4 as return val
```

(b) 汇编代码

图 3.8 算术运算函数体的 C 和汇编代码

省略了栈的建立和完成部分。

指令 3 实现表达式 x+y，一个操作数 y 来自寄存器%eax（由指令 1 取出），而另一个直接来自存储器。指令 4 和 5 执行计算 z*48，首先使 leal 指令对伸缩化的变址寻址模式的操作数执行计算：$(z+2z)=3z$，然后将这个值左移 4 位，以计算 $2^4 3z = 48z$。C 编译器常常用加法和移位指令来完成常数因子的乘法，就像 2.3.6 节中讨论的那样。指令 6 执行 AND 操作，而指令 7 执行最后的乘法。最后，指令 8 将返回值移到寄存器%eax。

在图 3.8 的汇编代码中，寄存器%eax 中的值先后对应于程序值 y、t1、t3 和 t4（作为返回值）。通常，编译器产生的代码中，会用一个寄存器存放多个程序值，还会在寄存器之间传送程序值。

练习题 3.6

在编写循环的代码中

```
for (i = 0; i < n; i++)
    v += i;
```

我们发现下面的汇编代码行：

```
xorl %edx,%edx
```

请解释为什么我们的C代码中没有EXCLUSIVE-OR（异或）运算符，这里却会有这样的指令。这条指令实现的是C程序中什么操作？

3.5.5 特殊的算术操作

图3.9描述的是支持产生两个32位数字的全64位乘积以及整数除法的指令。

指令	效果	描述
imull *S*	R[%edx]:R[%eax] ← *S*×R[%eax]	有符号全64位乘法
mull *S*	R[%edx]:R[%eax] ← *S*×R[%eax]	无符号全64位乘法
cltd *S*	R[%edx]:R[%eax] ← 符号扩展(R[%eax])	转换为四字
idivl *S*	R[%edx] ← R[%edx]:R[%eax] mod *S* R[%edx] ← R[%edx]:R[%eax]÷*S*	有符号除法
divl *S*	R[%edx] ← R[%edx]:R[%eax] mod *S* R[%edx] ← R[%edx]:R[%eax]÷*S*	无符号除法

图 3.9 特殊的算术操作

这些操作提供了有符号和无符号数的全64位乘法和除法。一对寄存器%edx和%eax组成一个64位的四字。

图3.7中列出的imull指令称为“双操作数”乘法指令。它从两个32位操作数产生一个32位乘积，实现了2.3.4和2.3.5节中描述的操作$*^{u}_{32}$和$*^{t}_{32}$。回想一下，当将乘积截取为32位时，无符号乘和二进制补码乘的位级行为是一样的。IA32还提供了两个不同的“单操作数”乘法指令，以计算两个32位值的全64位乘积——一个是无符号数乘法（mull），而另一个是二进制补码乘法（imull）。这两条指令都要求一个参数必须在寄存器%eax中，而另一个是作为指令的源操作数给出的。然后乘积存放在寄存器%edx（高32位）和%eax（低32位）中。注意，虽然imull这个名字可以用于两个不同的乘法操作，但是汇编器能够通过计算操作数的数目，分辨出是想用哪条指令。

让我们来看看这个例子，假设有符号数x和y存储在相对于%ebp偏移量为8和12的位置，我们希望将它们的全64位乘积作为8个字节存放在栈顶。代码是像下面这样的：

```
     x at %ebp+8, y at %ebp+12
1    movl 8(%ebp),%eax        Put x in %eax
2    imull 12(%ebp)           Multiply by y
3    pushl %edx               Push high-order 32 bits
4    pushl %eax               Push low-order 32 bits
```

注意我们将两个寄存器入栈的顺序，对小端法（little-endian）机器来说是对的，在这种机器中栈是向低地址方向增长的（也就是说，乘积的低位字节的地址比高位字节的地址小）。

我们前面的算术运算表（图3.7）没有列出除法或模（modulus）操作。单操作数除法类似于单

操作数乘法。有符号除法指令 idivl 将寄存器%edx（高 32 位）和%eax（低 32 位）中的 64 位数作为被除数，除数是作为指令的操作数给出的。指令将商存储在寄存器%eax 中，将余数存储在寄存器%edx 中。cltd[1] 指令可以用来根据寄存器%eax 中存放的 32 位的值形成 64 位被除数，这条指令将%eax 符号扩展到%edx。

让我们来看个例子，假设有符号数 x 和 y 存储在相对于%ebp 偏移量为 8 和 12 的位置，我们想要将 x/y 和 x%y 存储到栈中。代码是像下面这样的：

```
    x at %ebp+8, y at %ebp+12
1   movl 8(%ebp),%eax          Put x in %eax
2   cltd                       Sign extend into %edx
3   idivl 12(%ebp)             Divide by y
4   pushl %eax                 Push x / y
5   pushl %edx                 Push x % y
```

divl 指令执行无符号除法。通常会事先将寄存器%edx 设置为 0。

3.6 控制

到目前为止，我们考虑了访问数据和操作数据的方法。程序执行的另一个很重要的部分就是控制被执行操作的顺序。对 C 和汇编代码中的语句，默认的方式是顺序的控制流，按照语句或指令在程序中出现的顺序来执行。C 中的某些程序结构，比如条件语句、循环语句和分支语句，允许控制按照非顺序方式进行，即根据程序数据的值来确定顺序。

汇编代码提供了实现非顺序控制流的较低层次的机制。基本操作是跳转到程序的另一部分，可能会视某些测试结果而定。编译器产生的指令序列是依赖于这些低层机制来实现 C 的控制结构。

在我们的讲述中，会先谈到机器级机制，然后会给出如何用它们来实现 C 的各种控制结构。

3.6.1 条件码

除了整数寄存器，CPU 还包含一组单个位的条件码（condition code）寄存器，它们描述了最近的算术或逻辑操作的属性。对这些寄存器的检测，将有助于执行条件分支指令。最有用的条件码是：

CF：进位标志。最近的操作使最高位产生了进位，它可用来检查无符号操作数的溢出。

ZF：零标志。最近的操作得出的结果为 0。

SF：符号标志。最近的操作得到的结果为负数。

OF：溢出标志。最近的操作导致一个二进制补码溢出——正溢出或负溢出。

比如说，我们用 addl 指令完成等价于 C 表达式 t=a+b 的功能，这里变量 a、b 和 t 都是整型的。然后，会根据下面的表达式来设置条件码：

```
CF: (unsigned t) <(unsigned a)             无符号溢出
ZF: (t == 0)                               零
SF: (t < 0)                                负数
OF: (a < 0 == b < 0) && (t < 0 != a < 0)   有符号溢出
```

1 在 Intel 的文档里，这条指令称为 cdq。这是少数 GAS 指令名与 Intel 的名字无关的情况之一。

leal 指令不改变任何条件码，因为它是用来进行地址计算的。另一方面，图 3.7 中列出的所有指令都会设置条件码。对于逻辑操作，例如 xorl，进位标志和溢出标志会设置成 0。对于移位操作，进位标志将设置为最后一个被移出的位，而溢出标志设置为 0。

除了图 3.7 中的操作，下面的表给出了两个操作（有 8、16 和 32 位形式），它们只设置条件码而不改变任何其他寄存器。

指令		基于	描述
cmpb	S_2, S_1	$S_1 - S_2$	比较字节
testb	S_2, S_1	S_1 & S_2	测试字节
cmpw	S_2, S_1	$S_1 - S_2$	比较字
testw	S_2, S_1	S_1 & S_2	测试字
cmpl	S_2, S_1	$S_1 - S_2$	比较双字
testl	S_2, S_1	S_1 & S_2	测试双字

cmpb、cmpw 和 cmpl 指令根据它们的两个操作数之差来设置条件码。在 GAS 格式中，操作数的顺序是相反的，使得代码有点难读。如果两个操作数相等，这些指令会将零标志设置为 1，而其他的标志可以用来确定两个操作数之间的大小关系。

testb、testw 和 testl 指令会根据它们的两个操作数的与（AND）来设置零标志和负数标志。通常两个操作数是一样的（例如，`testl %eax, %eax` 用来检查%eax 是负数、零，还是正数），或其中的一个操作数是用来指示哪些位应该被测试的掩码。

3.6.2 访问条件码

两种最常用的访问条件码的方法不是直接读取它们，而是根据条件码的某个组合，设置一个整数寄存器或是执行一条件分支指令。图 3.10 中描述的是各种 set 指令根据条件码的某个组合，将一个字节设置为 0 或者 1。目的操作数是八个单字节寄存器元素（图 3.2）之一，或是存储一个字节的存储器位置。为了得到一个 32 位结果，我们必须对最高的 24 位清零。

一个 C 判定条件（例如 a < b）的典型指令序列如下所示：

```
    Note: a is in %edx, b is in %eax
1   cmpl %eax,%edx        Compare a:b
2   setl %al              Set low order byte of %eax to 0 or 1
3   movzbl %al,%eax       Set remaining bytes of %eax to 0
```

movzbl 指令用来清零三个高位字节。

某些底层的机器指令可能有多个名字，我们称之为"同义名（synonym）"。比如说，"setg"（表示"设置大于"）和"setnle"（表示"设置不小于等于"）指的就是同一条机器指令。编译器和反汇编器会随意决定使用哪个名字。

虽然所有的算术操作都会设置条件码，但是各个 set 命令的描述都适用于这样一种情况：执行比较指令，根据计算 t = a - b 设置条件码。例如，就 sete 来说，即"当相等时设置（Set when equal）"指令。当 a = b 时，会得到 t = 0，因此零标志置位就表示相等。

指令	同义名	效果	设置条件
sete D	setz	D←ZF	相等/零
setne D	setnz	D←~ZF	不等/非零
sets D		D←SF	负数
setns D		D←~SF	非负数
setg D	setnle	D←~(SF^OF)&~ZF	大于（有符号>）
setge D	setnl	D←~(SF^OF)	大于等于（有符号>=）
setl D	setnge	D←SF^OF	小于（有符号<）
setle D	setng	D←(SF^OF) $\vert$ ZF	小于等于（有符号<=）
seta D	setnbe	D←~CF &~ZF	超过（无符号>）
setae D	setnb	D←~CF	超过或相等（无符号>=）
setb D	setnae	D←CF	低于（无符号<）
setbe D	setna	D←CF &~ZF	低于或相等（无符号<=）

图3.10 set指令

每条指令根据条件码的某个组合，将一个字节设置为0或者1。有些指令有“同义名”，也就是，同一条机器指令有别的名字。

类似地，考虑用setl，即“当小于时设置（Set when less）”指令，测试一个有符号比较。当a和b是用二进制补码表示时，对于a<b，计算两者之差时，我们会有a-b<0。当没有溢出发生时，符号标志置位就表明a<b。当因为a-b是一个很大的正数，出现正溢出时，我们会得到t<0。当因为a-b是一个很小的负数，出现负溢出时，我们会得到t>0。无论是这两种情况中的哪一种，符号标志都表示的是真正的差的反。因此，溢出和符号位的异或测试的就是a<b。其他的有符号比较测试是基于SF^OF和ZF的其他组合。

对于无符号比较的测试，当无符号参数a和b的整数差是负数时，也就是当(unsigned)a<(unsigned)b时，cmpl指令会设置进位标志。因此，这些测试使用的是进位标志和零标志的组合。

练习题3.7

在下面的C代码中，我们用“____”替换了一些比较运算符，并且省略了强制类型转换中的数据类型。

```
1    char ctest(int a, int b, int c)
2    {
3      char t1 =         a __          b;
4      char t2 =         b __ (    )   a;
5      char t3 = (    ) c __ (    )   a;
6      char t4 = (    ) a __ (    )   c;
7      char t5 =         c __          b;
8      char t6 =         a __          0;
9      return t1 + t2 + t3 + t4 + t5 + t6;
10   }
```

对原始的C代码，GCC产生了下面这样的汇编代码：

```
1     movl 8(%ebp),%ecx        Get a
2     movl 12(%ebp),%esi       Get b
3     cmpl %esi,%ecx           Compare a:b
4     setl %al                 Compute t1
5     cmpl %ecx,%esi           Compare b:a
```

```
6    setb -1(%ebp)         Compute t2
7    cmpw %cx,16(%ebp)     Compare c:a
8    setge -2(%ebp)        Compute t3
9    movb %cl,%dl
10   cmpb 16(%ebp),%dl     Compare a:c
11   setne %bl             Compute t4
12   cmpl %esi,16(%ebp)    Compare c:b
13   setg -3(%ebp)         Compute t5
14   testl %ecx,%ecx       Test a
15   setg %dl              Compute t6
16   addb -1(%ebp),%al     Add t2 to t1
17   addb -2(%ebp),%al     Add t3 to t1
18   addb %bl,%al          Add t4 to t1
19   addb -3(%ebp),%al     Add t5 to t1
20   addb %dl,%al          Add t6 to t1
21   movsbl %al,%eax       Convert sum from char to int
```

根据这些汇编代码，填上C代码中缺失的部分（比较运算符号和强制类型转换符号）。

3.6.3 跳转指令和它们的编码

在正常执行的情况下，指令按照它们出现的顺序一条一条地执行。跳转（jump）指令会导致执行切换到程序中一个全新的位置（参见图3.11）。这些跳转的目的地通常用一个标号（label）指明。考虑下面这样的汇编代码序列：

```
1    xorl %eax,%eax       Set %eax to 0
2    jmp .L1              Goto .L1
3    movl (%eax),%edx     Null pointer dereference
4  .L1:
5    popl %edx
```

指　令	同义名	跳转条件	描　述
jmp *Label* jmp **Operand*		1 1	直接跳转 间接跳转
je *Label* jne *Label*	jz jnz	ZF ~ZF	相等/零 不相等/非零
js *Label* jns *Label*		SF ~SF	负数 非负数
jg *Label* jge *Label* jl *Label* jle *Label*	jnle jnl jnge jng	~(SF^OF)&~ZF ~(SF^OF) SF^OF (SF^OF)|ZF	大于（有符号>） 大于或等于（有符号>=） 小于（有符号<） 小于或等于（有符号<=）
ja *Label* jae *Label* jb *Label* jbe *Label*	jnbe jnb jnae jna	~CF&~ZF ~CF CF CF&~ZF	超过（无符号>） 超过或相等（无符号>=） 低于（无符号<） 低于或相等（无符号<=）

图3.11 jump指令

当跳转条件满足时，这些指令会跳转到一条带标号的目的地。有些指令有“同义名”，也就是同一条机器指令的别名。

指令 jmp .L1 会导致程序跳过 movl 指令，从 popl 指令开始继续执行。在产生目标代码文件时，汇编器会确定所有带标号指令的地址，并使跳转目标（目的指令的地址）编码为跳转指令的一部分。

jmp 指令是无条件跳转。它可以是直接跳转，即跳转目标是作为指令的一部分编码的，也可以是间接跳转，即跳转目标是从寄存器或存储器位置中读出的。汇编语言中，直接跳转是给出一个标号作为跳转目标的，例如，上面代码中的标号“.L1”。间接跳转的写法是“*”后面跟一个操作数指示符，语法与 movl 指令使用的一样。看看这个例子，指令

```
jmp *%eax
```

用寄存器%eax 中的值作为跳转目标，而指令

```
jmp *(%eax)
```

以%eax 中的值作为读地址，从存储器中读出跳转目标。

其他的跳转指令是根据条件码的某个组合，或者跳转，或者继续执行代码序列中下一条指令。请注意这些指令的名字和跳转条件与 set 指令是相匹配的。同 set 指令一样，一些底层的机器指令有多个名字。条件跳转只能是直接跳转。

虽然我们不关心目标代码格式的细节，但是理解跳转指令的目标是如何编码的对第 7 章中研究链接非常重要。此外，在解释反汇编器输出时，它也是很有帮助的。在汇编代码中，跳转目标是用符号标号书写的。汇编器，以及后来的链接器，会产生跳转目标的适当编码。跳转指令有几种不同的编码，但是最常用的一些是 PC 相关的（PC-relative，PC = Program Counter）。也就是，它们会将目标指令的地址与紧跟在跳转指令后面那条指令的地址之间的差作为编码。这些地址偏移量可以编码为一、二或四个字节。第二种编码方法是给出“绝对”地址，用四个字节直接指定目标。汇编器和链接器会选择适当的跳转目的编码。

作为一个与 PC 相关的寻址的例子，下面这个汇编代码的片断是编译 silly.c 文件所产生的。它包含两个跳转：第 1 行的 jle 指令前向跳转到更高的地址，而第 8 行的 jg 指令后向跳转到较低的地址。

```
1    jle .L4              If <=, goto dest2
2    .p2align 4,,7        Aligns next instruction to multiple of 8
3  .L5:                   dest1:
4    movl %edx,%eax
5    sarl $1,%eax
6    subl %eax,%edx
7    testl %edx,%edx
8    jg .L5               If >, goto dest1
9  .L4:                   dest2:
10   movl %edx,%eax
```

注意，第 2 行是一条针对汇编器的命令（directive），它会使后面指令的地址从 16 的倍数处开始，而最多浪费 7 个字节。这条命令是为了使处理器能更优化地使用指令高速缓存存储器（instruction cache memory）。

汇编器产生的“.o”格式的反汇编版本是这样的：

```
1    8: 7e 11               jle 1b <silly+0x1b>     Target = dest2
2    a: 8d b6 00 00 00 00   lea 0x0(%esi),%esi      Added nops
```

```
3      10: 89 d0                mov %edx,%eax          dest1:
4      12: c1 f8 01             sar $0x1,%eax
5      15: 29 c2                sub %eax,%edx
6      17: 85 d2                test %edx,%edx
7      19: 7f f5                jg 10 <silly+0x10>       Target = dest1
8      1b: 89 d0                mov %edx,%eax          dest2:
```

第2行的`lea 0x0(%esi),%esi`指令没有什么实际的效果。它是作为6个字节的空指令(nop)，使得下一条指令（第3行）的起始地址是16的倍数。

右边反汇编器产生的注释中，指令1的跳转目标明确指明为0x1b，指令7的是0x10。不过，观察指令的字节编码，会看到跳转指令1的目标编码（在第二个字节中）为0x11（十进制17）。把它加上0xa（十进制10），也就是下一条指令的地址，就得到跳转目标地址0x1b（十进制27），也就是指令8的地址。

类似地，跳转指令7的目标用单字节、二进制补码表示编码为0xf5（十进制-11）。将这个数加上0x1b（十进制27），即指令8的地址，我们得到0x10（十进制16），即指令3的地址。

正如这些例子说明的那样，当执行与PC相关的寻址时，程序计数器的值是跳转指令后面的那条指令的地址，而不是跳转指令本身的地址。这种惯例可以追述到早期的实现，当时，处理器会将更新程序计数器作为执行一条指令的第一步。

下面是链接后的程序反汇编的版本：

```
1   80483c8:  7e 11               jle    80483db <silly+0x1b>
2   80483ca:  8d b6 00 00 00 00   lea    0x0(%esi),%esi
3   80483d0:  89 d0               mov    %edx,%eax
4   80483d2:  c1 f8 01            sar    $0x1,%eax
5   80483d5:  29 c2               sub    %eax,%edx
6   80483d7:  85 d2               test   %edx,%edx
7   80483d9:  7f f5               jg     80483d0 <silly+0x10>
8   80483db:  89 d0               mov    %edx,%eax
```

这些指令被重定位到不同的地址，但是第1行和第7行中跳转目标的编码并没有变。通过使用与PC相关的跳转目标编码，指令编码很简洁（只需要两个字节），而且目标代码可以不做改变就移到存储器中不同的位置。

练习题 3.8

在下面这些反汇编二进制代码节选中，有些信息被X代替了。回答下列关于这些指令的问题：

A. 下面jbe指令的目标是什么？

```
8048d1c:  76 da            jbe    XXXXXXX
8048d1e:  eb 24            jmp    8048d44
```

B. mov指令的地址是多少？

```
XXXXXXX:  eb 54            jmp    8048d44
XXXXXXX:  c7 45 f8 10 00   mov    $0x10,0xfffffff8(%ebp)
```

C. 在下面的代码中，跳转目标的编码是PC相关的，且是一个4字节的二进制补码数。字节是

按照从最低位到最高位的顺序列出的，反映出IA32的小端法字节顺序。跳转目标的地址是什么？

```
8048902:  e9 cb 00 00 00 jmp  XXXXXXX
8048907:  90             nop
```

D. 请解释右边的注释与左边的字节代码之间的关系。这两行都是jmp指令编码的一部分。

```
80483f0:  ff 25 e0 a2 04     jmp   *0x804a2e0
80483f5:  08
```

为了实现C的控制结构，编译器必须使用刚才我们看到的各种类型的跳转指令。我们会浏览一下最常见的结构，从简单的条件分支开始，然后考虑循环和开关语句。

3.6.4 翻译条件分支

C中的条件语句是用有条件和无条件跳转结合起来实现的。例如，图3.12给出了一个计算两数之差绝对值的函数的C代码（a）。（c）是GCC产生的汇编代码，我们创建了对应的C版本，称为gotodiff（b），它是更加紧密地遵循汇编代码的控制流。（b）使用了C中的goto语句，这个语句类似于汇编代码中的无条件跳转。第6行的`goto less`语句会导致一个跳转，转移到第9行的标号less处，略过了第7行上的语句。请注意，通常认为使用goto语句是一种不好的编程风格，因为它会使代码难以阅读和调试。在我们的叙述中使用goto语句，是为了构造描述汇编代码程序控制流的C程序。我们称这样的C程序为"goto代码"。

汇编代码实现首先比较两个操作数（第3行），设置条件码。如果比较的结果表明x小于y，那么它就会跳转到计算y-x的代码块（第9行），否则就继续执行计算x-y的代码（第5行和第6行）。在这两种情况中，计算结果都存放在寄存器%eax中，到第10行结束，在此，它会执行栈完成代码（没有显示出来）。

C中的if-else语句的通用形式是这样的：

```
if  (test-expr)
  then-statement
else
  else-statement
```

这里test-expr是一个整数表达式，它的取值为0（解释为"假"）或者为非0（解释为"真"）。两个分支语句中（then-statement和else-statement）只会执行一个。

对于这种通用形式，汇编实现通常会使用下面这种形式，这里，我们用C语法来描述控制流：

```
    t = test-expr;
    if (t)
      goto true;
    else-statement
    goto done;
true:
    then-statement
done:
```

也就是，汇编器为then-statement和else-statement产生各自的代码块，并插入条件和无条件分支，以保证能执行正确的代码块。

code/asm/abs.c

```
1  int absdiff(int x, int y)
2  {
3      if (x < y)
4          return y - x;
5      else
6          return x - y;
7  }
```

code/asm/abs.c

（a）原始的C代码

code/asm/abs.c

```
1   int gotodiff(int x, int y)
2   {
3       int rval;
4
5       if (x < y)
6           goto less;
7       rval = x - y;
8           goto done;
9     less:
10          rval = y - x;
11    done:
12          return rval;
13  }
```

code/asm/abs.c

（b）与之等价的goto版本

```
1      movl 8(%ebp),%edx        Get x
2      movl 12(%ebp),%eax       Get y
3      cmpl %eax,%edx           Compare x:y
4      jl .L3                   If <, goto less
5      subl %eax,%edx           Compute x-y
6      movl %edx,%eax           Set as return value
7      jmp .L5                  Goto done
8    .L3:                     less:
9      subl %edx,%eax           Compute y-x as return value
10   .L5:                     done: Begin completion code
```

（c）产生的汇编代码

图 3.12 条件语句的编译

C 过程 absdiff（a）包含一个 if-else 语句，产生的汇编代码为（c），而 C 过程 gotodiff（b）模拟了汇编代码的控制流。注意：汇编代码中栈的建立和完成部分被省略。

练习题 3.9

当给出下列 C 代码时

code/asm/simple-if.c

```
1   void cond(int a, int *p)
2   {
3       if (p && a > 0)
4           *p += a;
5   }
```

code/asm/simple-if.c

GCC 会产生下面的汇编代码：

```
1	movl 8(%ebp),%edx
2	movl 12(%ebp),%eax
3	testl %eax,%eax
4	je .L3
5	testl %edx,%edx
6	jle .L3
7	addl %edx,(%eax)
8	.L3:
```

A. 按照图 3.12（b）中所示的风格，用 C 写一个 goto 版本，执行同样的计算，并模拟汇编代码的控制流。像我们在示例中那样给汇编代码加上注解可能会有帮助。

B. 请说明为什么 C 代码中只有一个 if 语句，而汇编代码包含两个条件分支。

3.6.5 循环

C 提供了好几种循环结构，即 while、for 和 do-while。汇编中没有相应的指令存在。作为替代，将条件测试和跳转组合起来实现循环的效果。有趣的是，大多数汇编器根据一个循环的 do-while 形式来产生循环代码，即使在实际程序中，这种形式用的相对较少。其他的循环会首先转换成 do-while 形式，然后编译成机器代码。我们会循序渐进地研究循环的翻译，从 do-while 开始，然后再研究更复杂的实现。

do-while 循环

do-while 语句的通用形式是这样的：

```
do
    body-statement
    while (test-expr);
```

循环的效果就是重复执行 body-statement，对 test-expr 求值，如果求值的结果为非零，就继续循环。注意，body-statement 至少执行一次。

通常，do-while 的实现有下面这样的通用形式：

```
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
```

作为一个示例，图 3.13 给出了一个用 do-while 循环计算 Fibonacci 序列中第 n 项的函数的实现。Fibonacci 序列是这样递归定义的：

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 3$$

比如说，该序列的前 10 个元素是 1、1、2、3、5、8、13、21、34 和 55。用 do-while 循环来实现，序列是从 $F_0 = 0$ 和 $F_1 = 1$ 开始，而不是从 F_1 和 F_2 开始的。

code/asm/fib.c

```
1  int fib_dw(int n)
2  {
3      int i = 0;
4      int val = 0;
5      int nval = 1;
6
7      do {
8          int t = val + nval;
9          val = nval;
10         nval = t;
11         i++;
12     } while (i < n);
13
14     return val;
15 }
```

code/asm/fib.c

(a) C 代码

寄存器用法		
寄存器	变量	初值
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	-

```
1  .L6:                              loop:
2    leal (%edx,%ebx),% eax          Compute t = val + nval
3    movl %edx,%ebx                  copy nval to val
4    movl %eax,%edx                  Copy t to nval
5    incl %ecx                       Increment i
6    cmpl %esi,%ecx                  Compare i:n
7    jl .L6                          If less, goto loop
8    movl %ebx,%eax                  Set val as return value
```

(b) 对应的汇编语言代码

图 3.13 Fibonacci 程序 do-while 版本的 C 和汇编代码

只有循环内的代码被显示。

图中还显示了实现这个循环的汇编代码，以及一张列出寄存器和程序值之间对应关系的表。在这个例子中，body-statement 是第 8~11 行，对 t、val 和 nval 赋值，并将 i 加 1。这些功能是由汇编代码的第 2~5 行实现的。表达式 i < n 就是 test-expr。第 6 行和第 7 行的跳转指令的测试条件实现了这个表达式。一旦退出循环，就会将 val 拷进寄存器%eax，作为返回值（第 8 行）。

创建一个像图 3.13（b）中那样的寄存器使用表，对于分析汇编语言程序是很有帮助的，特别是出现循环时。

练习题 3.10

对于 C 代码

```
1  int dw_loop(int x, int y, int n)
2  {
```

```
3   do {
4     x += n;
5     y *= n;
6     n--;
7   } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
8   return x;
9 }
```

GCC产生了下面这样的汇编代码：

```
    Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1    movl 8(%ebp),%esi
2    movl 12(%ebp),%ebx
3    movl 16(%ebp),%ecx
4    .p2align 4,,7              Inserted to optimize cache performance
5   .L6:
6    imull %ecx,%ebx
7    addl %ecx,%esi
8    decl %ecx
9    testl %ecx,%ecx
10   setg %al
11   cmpl %ecx,%ebx
12   setl %dl
13   andl %edx,%eax
14   testb $1,%al
15   jne .L6
```

A. 创建一个寄存器使用表，类似于图3.13（b）中所示的那个。

B. 指出C代码中的test-expr和body-statement，以及汇编代码中相应的行。

C. 对汇编代码添加一些注释，描述程序的操作，类似于图3.13（b）中所示的那样。

while 循环

while语句的通用形式是这样的：

while (*test-expr*)
    body-statement

它与do-while的不同之处在于对test-expr求值，在第一次执行body-statement之前，循环就可能中止了，直接翻译成使用goto语句的形式就是

```
loop:
  t = test-expr;
  if (!t)
    goto done;
  body-statement
  goto loop;
done:
```

这种翻译要求内循环，也就是执行次数最多的代码部分，里有两条控制语句。相反，大多数C

编译器将这段代码转换成do-while循环，用一个条件分支来在需要时省略循环体的第一次执行：

```
if (!test-expr)
  goto done;
do
  body-statement
  while (test-expr);
done:
```

然后，这段代码可以转换成带goto语句的代码：

```
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
done:
```

作为一个例子，图3.14给出了一个用while循环来实现Fibonacci序列函数的实现（a）。注意，这次我们的递归从元素F_1(val)和F_2(nval)开始。旁边的C函数fib_w_goto（b）表明了这段代码是如何翻译成汇编的，而（c）中的汇编代码非常接近于fib_w_goto中的C代码。编译器进行了几个非常有趣的优化，可以在goto代码（b）中看到。首先，编译器不是使用变量i作为循环变量并且在每次重复时拿它与n做比较，而是引入了一个新的称为“nmi”的循环变量，与原来的代码相比，它的值等于$n-i$。这使得编译器只用三个寄存器作为循环变量，而不用四个。其次，它将最原始的测试条件（$i<n$）优化成了（val $<n$），因为i和val的初始值都是1。这样一来，编译器就能完全消除变量i了。编译器常常利用变量的初始值来优化初始的测试，不过这使得解读汇编代码有点麻烦。第三，为了循环的连续执行，要保证$i \le n$，这样编译器就能假设nmi是非负的了。因此，它就能将nmi != 0而不是nmi >= 0作为循环条件来测试了。这样就在汇编代码中省略了一条指令。

练习题 3.11

对于下面的C代码：

```
1   int loop_while(int a, int b)
2   {
3     int i = 0;
4     int result = a;
5     while (i < 256) {
6       result += a;
7       a -= b;
8       i += b;
9     }
10    return result;
11  }
```

GCC 产生这样的汇编代码：

```
   Initially a and b are at offsets 8 and 12 from %ebp
1     movl 8(%ebp),%eax
2     movl 12(%ebp),%ebx
3     xorl %ecx,%ecx
4     movl %eax,%edx
5     .p2align 4,,7
6    .L5:
7     addl %eax,%edx
8     subl %ebx,%eax
9     addl %ebx,%ecx
10    cmpl $255,%ecx
11    jle .L5
```

A. 创建一个循环体内的寄存器使用表，类似于图 3.14（c）中所示的那一个。

B. 指出 C 代码中的 test-expr 和 body-statement，以及汇编代码中相应的行。C 编译器对初始测试进行了什么优化？

C. 对汇编代码添加一些注释，描述程序的操作，类似于图 3.14（c）中所示的那样。

D.（用 C 语言）写一个该函数的 goto 版本，它的结构类似于汇编代码的结构，就像图 3.14（b）中所做的那样。

code/asm/fib.c

```
1  int fib_w(int n)
2  {
3      int i = 1;
4      int val = 1;
5      int nval = 1;
6
7      while (i < n) {
8          int t = val+nval;
9          val = nval;
10         nval = t;
11         i++;
12     }
13
14     return val;
15 }
```

code/asm/fib.c

（a）C 代码

code/asm/fib.c

```
1  int fib_w_goto(int n)
2  {
3      int val = 1;
4      int nval = 1;
5      int nmi, t;
6
7      if (val >= n)
8          goto done;
9      nmi = n-1;
10
11     loop:
12         t = val+nval;
13     val = nval;
14         nval = t;
15         nmi--;
16         if (nmi)
17             goto loop;
18
19   done:
20       return val;
21 }
```

code/asm/fib.c

（b）与之等价的goto版本

寄存器用法		
寄存器	变量	初值
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1

```
1     movl 8(%ebp),%eax        Get n
2     movl $1,%ebx             Set val to 1
3     movl $1,%ecx             Set nval to 1
4     cmpl %eax,%ebx           Compare val:n
5     jge .L9                  If >= goto done
6     leal -1(%eax),%edx       nmi = n-1
7   .L10:                    loop:
8     leal (%ecx,%ebx),%eax    Compute t = nval+val
9     movl %ecx,%ebx           Set val to nval
10    movl %eax,%ecx           Set nval to t
11    decl %edx                Decrement nmi
12    jnz .L10                 if != 0, goto loop
13  .L9:                     done:
```

(c) 对应的汇编语言代码

图 3.14 Fibonacci 的 while 版本的 C 和汇编代码

编译器进行了一些优化，包括用一个我们称为 nmi 的变量代替变量 i 的值。

for 循环

for 循环的通用形式是这样的：

```
for (init-expr; test-expr; update-expr)
  body-statement
```

C 语言标准说明，这样一个循环的行为与下面这段使用 while 循环的代码的行为一样：

```
init-expr;
while (test-expr) {
  body-statement
  update-expr;
}
```

也就是，程序首先会对初始表达式 init-expr 求值。然后进入循环，它会先对测试条件 test-expr 求值，如果测试结果为“假”就会退出，然后执行循环体 body-statement，最后对更新表达式 update-expr 求值。

这段代码编译后的形式是基于前面讲过的从 while 到 do-while 的转换的，首先给出 do-while 形式：

```
init-expr;
if (!test-expr)
  goto done;
do {
  body-statement
  update-expr;
} while (test-expr);
done:
```

然后，将它转换成goto代码：

```
  init-expr;
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  update-expr;
  t = test-expr;
  if (t)
    goto loop;
done:
```

作为一个示例，下面这段代码给出了一个使用for循环的Fibonacci函数的实现：

code/asm/fib.c

```
1   int fib_f(int n)
2   {
3       int i;
4       int val = 1;
5       int nval = 1;
6
7       for (i = 1; i < n; i++) {
8           int t = val+nval;
9           val = nval;
10          nval = t;
11      }
12
13      return val;
14  }
```

code/asm/fib.c

将这段代码转换成while循环形式得到的代码与图3.14中给出的函数fib_w的代码一样。实际上，GCC对两个函数产生的汇编代码就是一样的。

练习题 3.12

考虑下面的汇编代码：

Initially x, y, and n are offsets 8, 12, and 16 from %ebp

```
1      movl 8(%ebp),%ebx
2      movl 16(%ebp),%edx
3      xorl %eax,%eax
4      decl %edx
5      js .L4
6      movl %ebx,%ecx
7      imull 12(%ebp),%ecx
```

```
8      .p2align 4,,7        Inserted to optimize cache performance
9    .L6:
10     addl %ecx,%eax
11     subl %ebx,%edx
12     jns .L6
13   .L4:
```

前面的代码是编译下面形式的C代码得到的：

```
1    int loop(int x, int y, int n)
2    {
3      int result = 0;
4      int i;
5      for (i = ____; i ____ ; i = ___ ) {
6        result += _____ ;
7      }
8      return result;
9    }
```

你的任务就是填写C代码中缺失的部分，使该程序等价于生成的汇编代码。回忆一下，函数的结果是放在寄存器%eax中返回的。为了解决这个问题，你可能需要对寄存器的使用进行一点猜测，然后看看这些猜测是否合理。

A. 程序值result和i应该放在哪些寄存器中？

B. i的初始值是多少？

C. i的测试条件是什么？

D. 是如何更新i的？

E. 描述如何在循环体内增加result的C表达式，不会在一次循环到下一次循环之间改变其值。编译器发现了这个情况，将它的计算移到了循环之前。这个表达式是什么？

F. 填写出C代码缺失的部分。

3.6.6 switch语句

switch（开关）语句提供了根据一个整数索引值进行多重分支（multiway branching）的能力。在处理具有多种可能结果的测试时，这种语句特别有用。它们不仅提高了C代码的可读性，而且通过使用一种称为**跳转表**（jump table）的数据结构使得实现更加高效。跳转表是一个数组，表项 i 是一个代码段的地址，这个代码段实现的是当开关索引值等于 i 时程序应该采取的动作。程序代码用开关索引值来执行一个跳转表内的数组引用，确定跳转指令的目标。和使用一组很长的if-else语句相比，使用跳转表的优点是执行开关语句的时间与开关情况（switch cases）的数量无关。GCC根据开关情况的数量和开关情况值的稀少程度（sparsity）来翻译开关语句。当开关情况数量比较多（例如，四个或更多），并且值的范围跨度比较小时，就会使用跳转表。

图3.15（a）给出了一个C switch语句的示例。这个例子有些非常有意思的特征，包括情况标号（case labels）是不连续的（对于情况101和105是没有标号的），有些情况有多个标号（情况104和106），而有些情况则会落入其他情况（情况102），因为对应该情况的代码段没有以break语句结尾。

code/asm/switch.c
```
1   int switch_eg(int x)
2   {
3       int result = x;
4
5       switch (x) {
6
7       case 100:
8           result *= 13;
9           break;
10
11      case 102:
12          result += 10;
13          /* Fall through */
14
15      case 103:
16          result += 11;
17      break;
18
19      case 104:
20      case 106:
21          result *= result;
22          break;
23
24      default:
25          result = 0;
26      }
27
28      return result;
29  }
```
code/asm/switch.c

(a) switch 语句

code/asm/switch.c
```
1   /* Next line is not legal C */
2   code *jt[7] = {
3       loc_A, loc_def, loc_B, loc_C,
4       loc_D, loc_def, loc_D
5   };
6
7   int switch_eg_impl(int x)
8   {
9       unsigned xi = x - 100;
10      int result = x;
11
12      if (xi > 6)
13          goto loc_def;
14
15      /* Next goto is not legal C */
16      goto jt[xi];
17
18  loc_A: /* Case 100 */
19      result *= 13;
20      goto done;
21
22  loc_B: /* Case 102 */
23      result += 10;
24      /* Fall through */
25
26  loc_C: /* Case 103 */
27      result += 11;
28      goto done;
29
30  loc_D: /* Cases 104, 106 */
31      result *= result;
32      goto done;
33
34  loc_def: /* Default case */
35      result = 0;
36
37  done:
38      return result;
39  }
```
code/asm/switch.c

(b) 到扩展C的翻译

图 3.15 switch 语句示例以及到扩展 C 的翻译

到扩展 C（extended C）的翻译给出了跳转表 jt 的结构，以及是如何访问它的。实际上 C 中是不允许这样的表和访问的。

图 3.16 是编译 switch_eg 时产生的汇编代码。这段代码的行为用 C 的扩展形式来描述就是图 3.15（b）中的过程 switch_eg_impl。我们说“扩展的”是因为 C 本身并不提供支持这种跳转表所需的结

构，因此我们的代码并不是合法的C。数组jt包含7个表项，每个都是一个代码块的地址。为此，我们扩展了C，增加了数据类型code。

```
    Set up the jump table access
1     leal -100(%edx),%eax          Compute xi = x-100
2     cmpl $6,%eax                  Compare xi:6
3     ja .L9                        if >, goto loc_def
4     jmp *.L10(,%eax,4)            Goto jt[xi]

    Case 100
5   .L4:                            loc_A:
6     leal (%edx,%edx,2),%eax         Compute 3*x
7     leal (%edx,%eax,4),%edx         Compute x+4*3*x
8     jmp .L3                         Goto done

    Case 102
9   .L5:                            loc_B:
10    addl $10,%edx                   result += 10, Fall through

    Case 103
11  .L6:                            oc_C:
12    addl $11,%edx                 result += 11
13    jmp .L3                         Goto done

    Cases 104, 106
14  .L8:                            loc_D:
15    imull %edx,%edx                 result *= result
16    jmp .L3                         Goto done

    Default case
17  .L9:                            loc_def:
18    xorl %edx,%edx                  result = 0

    Return result
19  .L3:                            done:
20    movl %edx,%eax                  Set result as return value
```

图3.16 图3.15中switch语句示例的汇编代码

第1~4行建立起了跳转表的入口。为了保证当x的值小于100或大于106时会执行default开关情况指定的计算，代码生成了一个等于x-100的无符号值xi。对于介于100~106之间的x的值，xi的值在0~6之间，因为x-100的负值会绕回成非常大的无符号数。因此，当xi大于6时，代码用ja（无符号大于）指令来跳转到默认开关情况的代码。用jt来指向跳转表，代码会执行一个跳转，转移到表中表项xi处的地址。注意，这种形式的goto不是合法的C语句。指令4实现的是到跳转表中某个表项的转移。因为是间接跳转，目标是从存储器中读出的。读的有效地址是由标号.L10指

定的基地址加上变量 xi（放在寄存器%eax 中）的伸缩值（伸缩因子值为 4，因为跳转表的每个表项都是 4 个字节）确定的。

在汇编代码中，跳转表是用下面这样的声明表示的，我们添加了一些注释：

```
1   .section .rodata
2     .align 4            Align address to multiple of 4
3   .L10:
4     .long .L4           Case 100: loc_A
5     .long .L9           Case 101: loc_def
6     .long .L5           Case 102: loc_B
7     .long .L6           Case 103: loc_C
8     .long .L8           Case 104: loc_D
9     .long .L9           Case 105: loc_def
10    .long .L8           Case 106: loc_D
```

这些声明表明，在叫做“.rodata”（表示“只读数据”，“Read-Only Data”）的目标代码文件的段中，应该有一组 7 个“长”字（4 个字节），每个字的值都是与指定的汇编代码标号（例如，.L4）相关的指令地址。标号.L10 标志着这段分配的起始。与这个标号相对应的地址会作为间接跳转（指令 4）的基地址。

在 switch_eg_impl 中（图 3.15（b）），从标号 loc_A 开始，一直到 loc_D 和 loc_def 的代码块，实现了 switch 语句的五个不同的分支。可以观察到，当 x 超出 100～106 范围时（初始范围检查），或者当它等于 101 或 105 时（根据跳转表），都会执行标号为 loc_def 的代码块。注意标号为 loc_B 的代码块是如何落入标号为 loc_C 的代码块的。

练习题 3.13

在下面的 C 函数中，我们省略了 switch 语句的主体。在 C 代码中，开关情况标号（case labels）是不连续的，而有些情况还有多个标号。

```
int switch2(int x) {
  int result = 0;
  switch (x) {
    /* Body of switch statement omitted */
  }
  return result;
}
```

在编译函数时，GCC 为程序的初始部分以及跳转表生成了下面这样的汇编代码。变量 x 开始时是位于相对于寄存器%ebp 偏移量为 8 的地方。

```
    Setting up jump table access
1   movl 8(%ebp),%eax   Retrieve x
2   addl $2,%eax
3   cmpl $6,%eax
4   ja .L10
```

```
    Jump table for switch2
1   .L11:
2   .long .L4
3   .long .L10
4   .long .L5
5   .long .L6
6   .long .L8
7   .long .L8
```

```
5    jmp *.L11(,%eax,4)                8    .long .L9
```

根据前面的信息来回答下列问题：

A. switch 语句体内的开关情况标号的值是多少？

B. C 代码中哪些开关情况有多个标号？

3.7 过程

一个过程调用包括将数据（以过程参数和返回值的形式）和控制从代码的一部分传递到另一部分。另外，它还必须在进入时为过程的局部变量分配空间，并在退出时释放这些空间。大多数机器，包括 IA32，只提供简单的转移控制到过程和从过程中转移出控制的指令。数据传递、局部变量的分配和释放是通过操纵程序栈来实现的。

3.7.1 栈帧结构

IA32 程序用程序栈来支持过程调用。栈用来传递过程参数、存储返回信息、保存寄存器以供以后恢复之用，以及用于本地存储。为单个过程分配的那部分栈称为栈帧（stack frame）。图 3.17 描绘了栈帧的通用结构。栈帧的最顶端是以两个指针定界的，寄存器%ebp 作为帧指针，而寄存器%esp 作为栈指针。当程序执行时，栈指针是可以移动的，因此大多数信息的访问都是相对于帧指针的。

假设过程 P（调用者）调用过程 Q（被调用者）。Q 的参数放在 P 的栈帧中。另外，当 P 调用 Q 时，P 中的返回地址被压入栈中，形成 P 的栈帧的末尾，返回地址就是当程序从 Q 返回时应该继续执行的地方。Q 的栈帧从保存的帧指针的值（例如，%ebp）开始，后面是保存的其他寄存器的值。

过程 Q 也用栈来保存其他不能存放在寄存器中的局部变量。这样做是因为：

- 寄存器不够存放所有的局部变量。
- 有些局部变量是数组或结构，因此必须通过数组或结构引用来访问。
- 要对一个局部变量使用地址操作符“&”，因此我们必须能够为它产生一个地址。

最后，Q 会用栈帧来存放它调用其他过程的参数。

正如前面讲过的那样，栈向低地址方向增长，而栈指针%esp 指向栈顶元素。可以通过 pushl 和 popl 指令将数据存入栈中和从栈中取出。可以通过将栈指针的值减小适当的值来分配没有指定初始值的数据的空间。类似地，可以通过增加栈指针来释放空间。

3.7.2 转移控制

下表给出的是支持过程调用和返回的指令：

指　令	描　述
`call` *Label*	过程调用
`call` **Operand*	过程调用
`leave`	为返回准备栈
`ret`	从过程调用中返回

call 指令有一个目标，指明被调用过程起始的指令地址。同跳转一样，调用可以是直接的，也可以是间接的。在汇编代码中，直接调用的目标是一个标号，而间接调用的目标是*后面跟一个操作

数指示符，其语法与 movl 指令的操作数的语法相同（图 3.3）。

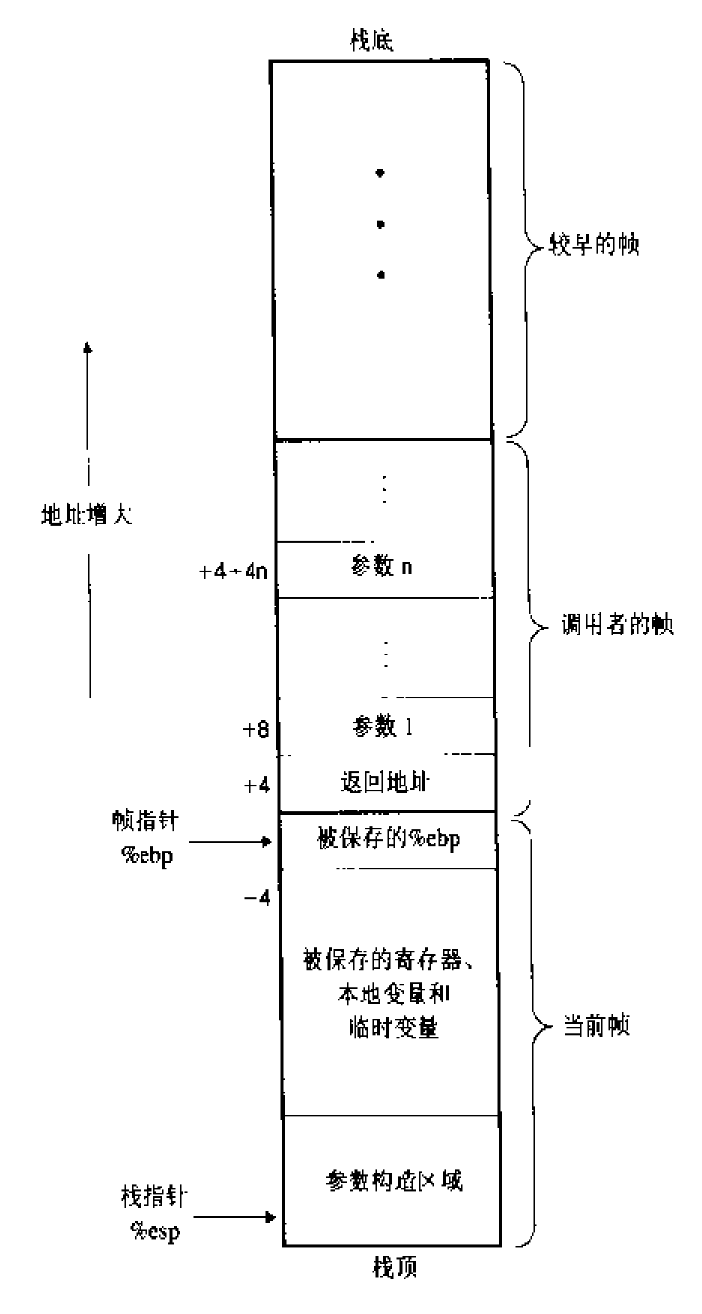


图 3.17 栈帧结构

栈用来传递参数、存储返回信息、保存寄存器，以及用于本地存储。

call 指令的效果是将返回地址入栈，并跳转到被调用过程的起始处。返回地址是紧跟在程序中 call 后面的那条指令的地址，这样当被调用过程返回时，执行会从此继续。ret 指令从栈中弹出地址，并跳转到那个位置。要正确使用这条指令，就要使栈准备好，栈指针要指向前面 call 指令存储返回地址的位置。leave 指令可以用来使栈做好返回的准备。它等价于下面的代码序列：

```
1    movl %ebp, %esp    Set stack pointer to beginning of frame
2    popl %ebp          Restore saved %ebp and set stack ptr
                        to end of caller's frame
```

另外，这种准备工作也可以通过直接使用传送和弹出操作来完成。

寄存器%eax可以用来返回值，如果函数要返回整数或指针的话。

练习题 3.14

下面的代码片断常常出现在库函数的编译版本中：

```
1        call next
2    next:
3        popl %eax
```

A. 寄存器%eax设置成了什么值？

B. 解释为什么这个调用没有匹配的ret指令。

C. 这段代码完成了什么功能？

3.7.3 寄存器使用惯例

程序寄存器组是惟一一个被所有过程共享的资源。虽然在给定时刻只能有一个过程是活动的，我们必须保证当一个过程（调用者）调用另一个（被调用者）时，被调用者不会覆盖某个调用者稍后会使用的寄存器的值。为此，IA32采用了一组统一的寄存器使用惯例，所有的过程都必须遵守，包括程序库中的过程。

根据惯例，寄存器%eax、%edx和%ecx被划分为调用者保存（caller save）寄存器。当过程P调用Q时，Q可以覆盖这些寄存器，而不会破坏任何P所需要的数据。另外，寄存器%ebx、%esi和%edi被划分为被调用者保存（callee save）寄存器。这意味着Q必须在覆盖它们之前，将这些寄存器的值保存到栈中，并在返回前恢复它们，因为P（或某个更高层次的过程）可能会在今后的计算中需要这些值。此外，根据这里描述的惯例，必须保持寄存器%ebp和%esp。

旁注：为什么叫做“被调用者保存”和“调用者保存”？

考虑下面这个场景：

```
int P()
{
    int x = f();    /* Some computation */
    Q();
    return x;
}
```

过程P希望它计算出来的x的值在调用了Q之后仍然有效。如果x放在一个调用者保存寄存器中，而P（调用者）必须在调用Q之前保存这个值，并在Q返回后恢复该值。如果x在一个被调用者保存寄存器中，Q（被调用者）想使用这个寄存器，那么Q在使用这个寄存器之前，必须保存这个值，并在返回前恢复它。在这两种情况中，保存就是将寄存器值压入栈中，而恢复是指从栈中弹出到寄存器中。

作为一个示例，考虑下面这段代码：

```
1  int P(int x)
2  {
3      int y = x*x;
```

```
4     int z = Q(y);
5
6     return y + z;
7 }
```

过程P在调用Q之前计算y，但是它必须保证y的值在Q返回后是可用的。有两种方式可以做到这一点：

- 它可以在调用Q之前，将y的值存放在自己的栈帧中；当Q返回时，它可以从栈中取出y的值。
- 它可以将y的值保存在被调用者保存寄存器中。如果Q，或任何其他Q调用的程序，想使用这个寄存器，它必须将这个寄存器的值保存在栈帧中，并在返回前恢复该值。因此，当Q返回到P时，y的值会在被调用者保存寄存器中，或者是因为寄存器根本就没有改变，或者是因为它被保存并恢复了。

最常见的是，GCC使用后一种方法，因为它会尽量减少写和读栈的次数。

练习题 3.15

下面这段代码出现在GCC为一个C过程产生的汇编代码的前部：

```
1     pushl %edi
2     pushl %esi
3     pushl %ebx
4     movl 24(%ebp),%eax
5     imull 16(%ebp),%eax
6     movl 24(%ebp),%ebx
7     leal 0(,%eax,4),%ecx
8     addl 8(%ebp),%ecx
9     movl %ebx,%edx
```

我们看到，只将三个寄存器（%edi、%esi 和%ebx）保存到了栈中。然后程序会修改它们，以及另外三个寄存器（%eax、%ecx 和%edx）。过程结尾，用 popl 指令恢复寄存器%edi、%esi 和%ebx，而其他三个寄存器就保持修改过的状态。

请解释这种在保存和恢复寄存器状态中明显的矛盾。

3.7.4 过程示例

作为一个示例，考虑图3.18中定义的C过程。图3.19给出了这两个过程的栈帧。注意，swap_add从caller的栈帧中取出它的参数。这些参数的位置的访问都是相对于寄存器%ebp中的帧指针的。帧左边的数字表示相对于帧指针的地址偏移。

code/asm/swapadd.c

```
1     int swap_add(int *xp, int *yp)
2     {
3         int x = *xp;
4         int y = *yp;
5
6         *xp = y;
```

```
7       *yp = x;
8       return x + y;
9   }
10
11  int caller()
12  {
13      int arg1 = 534;
14      int arg2 = 1057;
15      int sum = swap_add(&arg1, &arg2);
16      int diff = arg1 - arg2;
17
18      return sum * diff;
19  }
```

code/asm/swapadd.c

图 3.18　过程定义和调用的示例

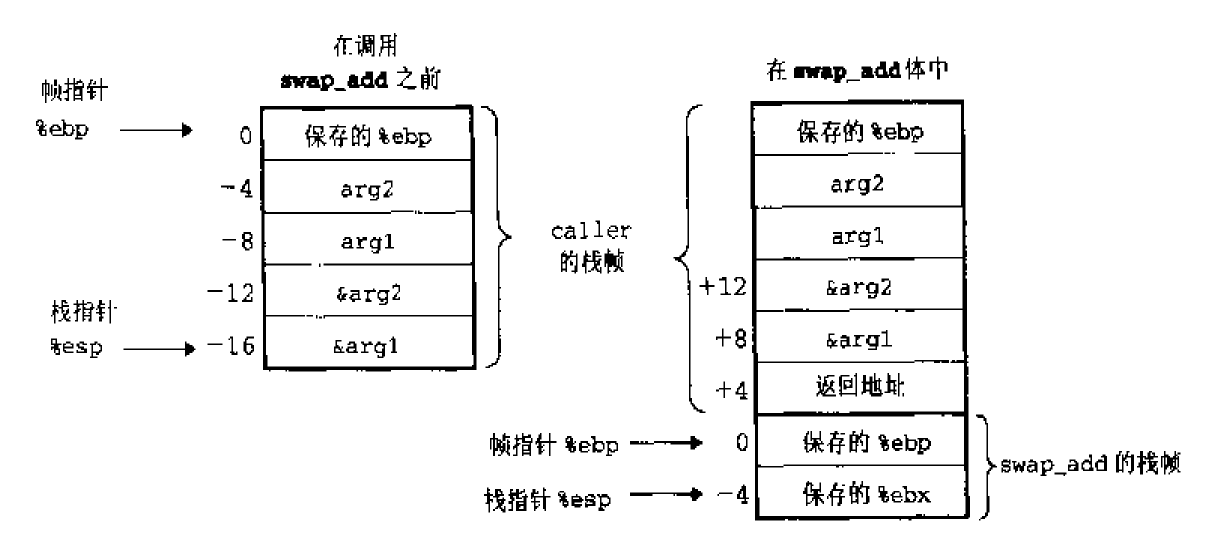


图 3.19　caller 和 swap_add 的栈帧

过程 swap_add 从 caller 的栈帧中取出它的参数。

caller 的栈帧包括局部变量 arg1 和 arg2 的存储，其位置相对于帧指针是-8 和-4。这些变量必须存在栈中，因为我们必须为它们产生地址。接下来的这段来自 caller 编译过的汇编代码显示出它是如何调用 swap_add 的。

```
    Calling code in caller
1   leal -4(%ebp),%eax        Compute &arg2
2   pushl %eax                Push &arg2
3   leal -8(%ebp),%eax        Compute &arg1
4   pushl %eax                Push &arg1
5   call swap_add             Call the swap_add function
```

注意，这段代码计算的是局部变量 arg2 和 arg1 的地址（用 leal 指令），并将它们压入栈中。然后再调用 swap_add。

swap_add 编译过的代码有三个部分："建立"部分，初始化栈帧；"主体"部分，执行过程的实际计算；"结尾"部分，恢复栈的状态和过程返回。

下面是 swap_add 的建立代码。回想一下，call 指令已经将返回地址压入栈中。

Setup code in swap_add

```
1    swap_add:
2      pushl %ebp              Save old %ebp
3      movl %esp,%ebp          Set %ebp as frame pointer
4      pushl %ebx              Save %ebx
```

过程 swap_add 需要用寄存器%ebx 作为临时存储。因为这是一个被调用者保存的寄存器，它会将旧值作为栈帧建立的一部分压入栈中。

下面是 swap_add 的主体代码：

Body code in swap_add

```
5     movl 8(%ebp),%edx        Get xp
6     movl 12(%ebp),%ecx       Get yp
7     movl (%edx),%ebx         Get x
8     movl (%ecx),%eax         Get y
9     movl %eax,(%edx)         Store y at *xp
10    movl %ebx,(%ecx)         Store x at *yp
11    addl %ebx,%eax Set       return value = x+y
```

这段代码从 caller 的栈帧中取出它的参数。因为帧指针已经移动了，这些参数的位置已经从相对于%ebp 的旧值的位置-12 和-6 移到了相对于%ebp 的新值的位置+12 和+8。注意，变量 x 和 y 的和是存放在寄存器%eax 中作为返回值传递的。

下面是 swap_add 的结尾代码：

Finishing code in swap_add

```
12    popl %ebx                Restore %ebx
13    movl %ebp,%esp           Restore %esp
14    popl %ebp                Restore %ebp
15    ret                      Return to caller
```

这段代码就是恢复三个寄存器%ebx、%esp 和%ebp 的值，然后执行 ret 指令。注意，可以用一条 leave 指令代替指令 13 和 14。不同版本的 GCC 对此可能会有不同的习惯。

下面的 caller 中的代码紧跟在调用 swap_add 的指令后面：

```
6    movl %eax,%edx            Resume here
```

从 swap_add 返回时，过程 caller 会从这条指令开始继续执行。注意，这条指令将返回值从%eax 拷贝到另一个寄存器。

练习题 3.16

给定一个 C 函数

```
1  int proc(void)
2  {
3    int x,y;
4    scanf("%x %x", &y, &x);
5    return x-y;
6  }
```

GCC 产生下面这样的汇编代码：

```
1   proc:
2     pushl %ebp
3     movl %esp,%ebp
4     subl $24,%esp
5     addl $-4,%esp
6     leal -4(%ebp),%eax
7     pushl %eax
8     leal -8(%ebp),%eax
9     pushl %eax
10    pushl $.LC0         Pointer to string "%x %x"
11    call scanf
      Diagram stack frame at this point
12    movl -8(%ebp),%eax
13    movl -4(%ebp),%edx
14    subl %eax,%edx
15    movl %edx,%eax
16    movl %ebp,%esp
17    popl %ebp
18    ret
```

假设过程 proc 开始执行时，寄存器的值如下：

寄存器	值
%ebp	0x800040
%ebp	0x800060

假设 proc 调用 scanf(第 11 行)，而 scanf 从标准输入读入值 0x46 和 0x53。假设字符串 "%x %x" 存放在存储器位置 0x300070。

A. 第 3 行上，%ebp 的值被设置成了多少？

B. 局部变量 x 和 y 的存放地址是什么？

C. 第 10 行后%esp 的值是多少？

D. 画出就在 scanf 返回后 proc 的栈帧的图，请包括尽可能多的关于栈帧元素的地址和内容的信息。

E. 指出 proc 未使用的栈帧区域（分配这些浪费了的区域是为了改进高速缓存的性能）。

3.7.5 递归过程

上一节中描述的栈和链接惯例使得过程能够递归地调用它们自身。因为每个调用在栈中都有它自己的私有空间，多个未完成调用的局部变量不会相互影响。此外，栈的原则很自然地就提供了适当的策略，当过程被调用时分配局部存储（storage），当返回时释放存储。

图 3.20 给出了递归的 Fibonacci 函数的 C 代码。（注意，这段代码的效率很低——我们用它来作为一个说明示例，这不是一个很聪明的算法。）完整的汇编代码如图 3.21 所示。

code/asm/fib.c

```
1   int fib_rec(int n)
2   {
3       int prev_val, val;
```

```
4
5        if (n <= 2)
6            return 1;
7        prev_val = fib_rec(n-2);
8        val = fib_rec(n-1);
9        return prev_val + val;
10   }
```

code/asm/fib.c

图 3.20 递归的 Fibonacci 程序的 C 代码

```
1     fib_rec:
        Setup code
2       pushl %ebp                 Save old %ebp
3       movl %esp,%ebp             Set %ebp as frame pointer
4       subl $16,%esp              Allocate 16 bytes on stack
5       pushl %esi                 Save %esi (offset -20)
6       pushl %ebx                 Save %ebx (offset -24)

        Body code
7       movl 8(%ebp),%ebx          Get n
8       cmpl $2,%ebx               Compare n:2
9       jle .L24                   if <=, goto terminate
10      addl $-12,%esp             Allocate 12 bytes on stack
11      leal -2(%ebx),%eax         Compute n-2
12      pushl %eax                 Push as argument
13      call fib_rec               Call fib_rec(n-2)
14      movl %eax,%esi             Store result in %esi
15      addl $-12,%esp             Allocate 12 bytes on stack
16      leal -1(%ebx),%eax         Compute n-1
17      pushl %eax                 Push as argument
18      call fib_rec               Call fib_rec(n-1)
19      addl %esi,%eax             Compute val+nval
20      jmp .L25                   Goto done

        Terminal condition
21    .L24:                      terminate:
22      movl $1,%eax               Return value 1

        Finishing code
23    .L25:                      done:
24      leal -24(%ebp),%esp        Set stack to offset -24
25      popl %ebx                  Restore %ebx
26      popl %esi                  Restore %esi
27      movl %ebp,%esp             Restore stack pointer
28      popl %ebp                  Restore %ebp
29      ret                        Return
```

图 3.21 图 3.20 中递归的 Fibonacci 程序的汇编代码

虽然代码有点长，但还是值得仔细研究一下的。建立代码（第 2～6 行）创建一个栈帧，其中包

含%ebp 的旧值、未使用的 16 个字节[2]、保存的被调用者保存寄存器%esi 和%ebx 的值，如图 3.22 左边所示。然后，它用寄存器%ebx 来保存过程参数 n（第 7 行）。一旦满足中止条件，代码会跳转到第 22 行，在此将返回值设为 1。

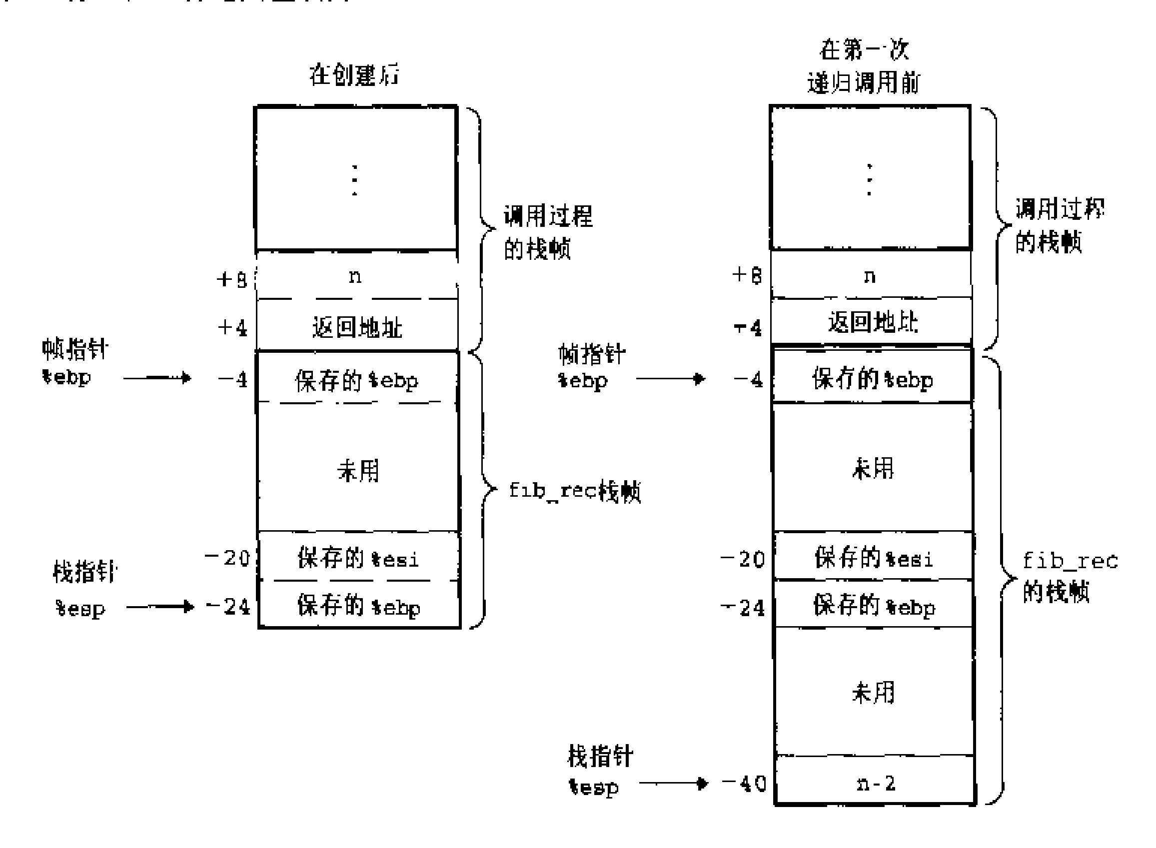


图 3.22 递归的 Fibonacci 函数的栈帧

左边是初始建立之后的帧状态；右边是第一次递归调用之前的帧状态。

对于不满足中止条件的情况，指令 10～12 会进行第一次递归调用。这包括在栈中分配不会被使用的 12 个字节，然后将计算出来的值 n-2 压入栈中。此时，栈帧如图 3.22 右边所示。然后，它会进行递归调用，引起一连串的调用、分配栈帧、对局部存储进行操作，等等。每次调用返回时，它都会释放栈空间，恢复所有被修改过的被调用者保存寄存器。因此，当我们返回到当前调用时（第 14 行），我们可以假设寄存器%eax 包含着递归调用返回的值，而寄存器%ebx 包含函数参数 n 的值。返回值（C 代码中的局部变量 prev_val）存放在寄存器%esi 中（第 14 行）。通过使用被调用者保存寄存器，我们能保证在第二次递归调用后这个值仍然是可用的。

指令 15～17 进行第二次递归调用。它会再次分配不会被使用的 12 个字节，并将值 n-1 压入栈中。在这个调用之后（第 18 行），计算出来的结果会放在寄存器%eax 中，而我们假设前一次调用的结果放在寄存器%esi 中。两者相加得到返回值（第 19 行）。

完成代码恢复寄存器和释放栈帧。它首先将栈帧设置为保存的%ebx 值的位置。注意，通过计算相对于%ebp 值的栈的位置，无论是否满足中止条件，计算都会是正确的。

2 不清楚为什么 C 编译器会为这个函数在栈中分配这么多的未使用存储（storage）。

3.8 数组分配和访问

C中数组是一种将标量型数据聚集成更大数据类型的方式。C用来实现数组的方式非常简单，因此很容易翻译成机器代码。C的一个不同寻常的特点是可以对数组中的元素产生指针，并对这些指针进行运算。这些运算会在汇编代码中翻译成地址计算。

优化编译器非常善于简化数组索引所使用的地址计算，不过这使得C代码和它到机器代码的翻译之间的对应关系很难理解。

3.8.1 基本原则

对于数据类型 T 和整常数 N，声明

T A[N];

有两个效果。首先，它在存储器中分配了 $L \cdot N$ 字节的连续区域，这里 L 是数据类型 T 的大小（单位为字节）。我们用 x_A 来表示起始位置。其次，它引入了标识符A，A可以用来作为指向数组开头的指针。这个指针的值就是 x_A。可以用从 $0 \sim N$-1之间的整数索引来访问数组元素。数组元素i将存放地址为 $x_A + L \cdot i$。

作为示例，让我们来看看下面这样的声明：

```
char     A[12];
char    *B[8];
double   C[6];
double  *D[5];
```

这些声明会产生带下列参数的数组：

数组	元素大小	总大小	起始地址	元素 i
A	1	12	x_A	x_A+i
B	4	32	x_B	x_B+4i
C	8	48	x_C	x_C+8i
D	4	20	x_D	x_D+4i

数组A由12个单字节（char）元素组成。数组C由6个双精度浮点值组成，每个值需要8个字节。B和D都是指针数组，因此每个数组元素都是4个字节。

IA32的存储器引用指令被设计用来简化数组访问。例如，假设E是一个整数数组，而我们想计算E[i]，在此，E的地址存放在寄存器%edx中，而i存放在寄存器%ecx中。然后，指令

```
movl (%edx,%ecx,4),%eax
```

会执行地址计算 $x_E + 4i$，在该存储器位置执行读操作，并将结果存放在寄存器%eax中。提示：伸缩因子1、2、4和8适用于基本数据类型的大小。

练习题 3.17

考虑下面的声明：

```
short           S[7];
short          *T[3];
```

```
short          **U[6];
long double      V[8];
long double     *W[4];
```

填写下表，描述每个数组的元素大小、整个数组的大小以及元素 i 的地址：

数组	元素大小	整个数组的大小	起始地址	元素 i
S			x_S	
T			x_T	
U			x_U	
V			x_V	
W			x_W	

3.8.2 指针运算

C 允许对指针进行运算，而计算出来的值会根据该指针引用的数据类型的大小进行调整。也就是说，如果 p 是一个指向类型 T 的数据的指针，p 的值为 x_p，表达式 p+i 的值为 $x_p + L \cdot i$，这里 L 是数据类型 T 的大小。

单操作数的操作符&和*可以产生指针和间接引用指针。也就是，对于一个表示某个对象的表达式 Expr，&Expr 表示一个地址。对于表示一个地址的表达式 Addr-Expr，*Addr-Expr 表示该地址中的值。因此，表达式 Expr 与*&Expr 是等价的。可以对数组和指针应用数组下标操作，如数组引用 A[i]与表达式*(A+i)是一样的。它计算第 i 个数组元素的地址，然后访问这个存储器位置。

扩充一下我们前面的例子，假设整数数组 E 的起始地址和整数索引 i 分别存放在寄存器%edx 和%ecx 中。下面是一些与 E 有关的表达式。我们还给出了每个表达式的汇编代码实现，结果存放在寄存器%eax 中。

表达式	类型	值	汇编代码
E	int *	x_E	movl %edx, %eax
E[0]	int	$M[x_E]$	movl (%edx), %eax
E[i]	int	$M[x_E+4i]$	movl (%edx, %ecx, 4), %eax
&E[2]	int *	x_E+8	leal 8(%edx), %eax
E+i-1	int *	x_E+4i-4	leal -4(%edx, %ecx, 4), %eax
*(&E[i]+i)	int	$M[x_E+4i+4i]$	movl (%edx, %ecx), %eax
&E[i]-E	int	i	movl %ecx, %eax

在这些例子中，leal 指令用来产生地址，而 movl 用来引用存储器（除了在第一种情况中，那里它是拷贝一个地址）。最后一个例子表明我们可以计算同一个数据结构中的两个指针之差，结果值是除以数据类型大小后的值。

练习题 3.18

假设短整型数组 S 的地址和整数索引 i 分别存放在寄存器%edx 和%ecx 中。对下面每个表达式，给出它的类型、值表达式和汇编代码实现。如果结果是指针的话，要保存在寄存器%eax 中，如果是

短整数，就保存在寄存器元素%ax 中。

表达式	类型	值	汇编代码
S+1			
S[3]			
&S[i]			
S[4*i+1]			
S+i-5			

3.8.3 数组与循环

在循环代码内，对数组的引用通常有非常规则的模式，优化编译器会使用这些模式。例如，图 3.23（a）中所示的函数 decimal5，计算的是一个由 5 个十进制数字的数组表示的整数。在把这个函数转换成汇编代码的过程中，编译器产生的代码类似于图 3.23(b)(中的 C 函数 decimal5_opt。首先，它不会使用循环变量 i，而是用指针运算来依次遍历数组元素。它计算出最后一个数组元素的地址，并且把与这个地址的比较作为循环测试。最后，它能使用 do-while 循环，因为至少要执行一次循环体。

图 3.23（c）中所示的代码给出了一个进一步的优化，以避免使用整数乘法指令。特别地，它使用 leal（第 5 行）来计算 5*val 作为 val+4*val。然后，用伸缩因子值为 2 的 leal（第 7 行）使之扩展为 10*val。

code/asm/decimal5.c

```
1   int decimal5(int *x)
2   {
3       int i;
4       int val = 0;
5
6       for (i = 0; i < 5; i++)
7           val = (10 * val) + x[i];
8
9       return val;
10  }
```

code/asm/decimal5.c

(a) 原始的 C 代码

code/asm/decimal5.c

```
1   int decimal5_opt(int *x)
2   {
3       int val = 0;
4       int *xend = x + 4;
5
6       do {
```

```
7          val = (10 * val) + *x;
8          x++;
9      } while (x <= xend);
10
11     return val;
12 }
```

code/asm/decimal5.c

(b) 等价的指针代码

```
   Body code
1   movl 8(%ebp),%ecx           Get base addr of array x
2   xorl %eax,%eax              val = 0;
3   leal 16(%ecx),%ebx          xend = x+4 (16 bytes = 4 double words)
4  .L12:                      loop:
5   leal (%eax,%eax,4),%edx     Compute 5*val
6   movl (%ecx),%eax Compute *x
7   leal (%eax,%edx,2),%eax     Compute *x + 2*(5*val)
8   addl $4,%ecx x++
9   cmpl %ebx,%ecx              Compare x:xend
10  jbe .L12                    if <=, goto loop
```

(c) 相应的汇编代码

图 3.23 数组循环示例的 C 和汇编代码

编译器产生的代码类似于 decimal5_opt 中所示的指针代码。

旁注：为什么要避免使用整数乘法？

在较老的 IA32 处理器模型中，整数乘法指令要花费 30 个时钟周期，所以编译器要尽可能地避免使用它。而在大多数新近的处理器模型中，乘法指令只需要 3 个时钟周期，所以不一定会进行这样的优化了。

3.8.4 嵌套数组

即使是创建数组的数组时，数组分配和引用的通用原则也是有效的。例如，声明

```
int A[4][3];
```

等价于声明

```
typedef int row3_t[3];
row3_t A[4];
```

数据类型 row3_t 被定义成一个三个整数的数组。数组 A 包含有四个这样的元素，每个都需要 12 个字节来存放三个整数。所以，总的数组大小为 4·4·3=48 字节。

数组 A 还可以看成是一个 4 行 3 列的二维数组，从 A[0][0]到 A[3][2]。数组元素在存储器中是按照“行优先”的顺序排列的，这就意味着先是行 0 的所有元素，后面是行 1 的所有元素，依此类推。

元素	地址
A[0][0]	x_A
A[0][1]	x_A+4
A[0][2]	x_A+8
A[1][0]	x_A+12
A[1][1]	x_A+16
A[1][2]	x_A+20
A[2][0]	x_A+24
A[2][1]	x_A+28
A[2][2]	x_A+32
A[3][0]	x_A+36
A[3][1]	x_A+40
A[3][2]	x_A+44

这种排序方法是我们嵌套声明的结果。将 A 看成一个四元素数组，每个元素又是一个三个 int 的数组，我们先有 A[0]（也就是行 0），后面是 A[1]，依此类推。

要访问多维数组中的元素，编译器产生的代码要计算待访问元素的偏移，然后再用 movl 指令，以数组的起始作为基地址，偏移（可能需要乘以伸缩因子）作为索引。通常，对一个声明如下的数组：

```
T D[R][C];
```

数组元素 D[i][j]是位于存储器地址 $x_D + L(C \cdot i + j)$的，这里 L 是用字节表示的数据类型 T 的大小。

看看下面这个例子，考虑前面定义的 4 × 3 的整数数组。假设寄存器%eax 包含 x_A，%edx 保存着 i，而%ecx 保存着 j。然后，下面的代码将拷贝数组元素 A[i][j]到寄存器%eax：

```
      A in %eax, i in %edx, j in %ecx
1     sall  $2,%ecx                 j*4
2     leal  (%edx,%edx,2),%edx      i*3
3     leal  (%ecx,%edx,4),%edx      j*4 + i*12
4     movl  (%eax,%edx),%eax        Read M[x_A + 4(3·i + j)]
```

练习题 3.19

考虑下面的源代码，此处，M 和 N 是用#define 声明的常数：

```
#define:
1   int mat1[M][N];
2   int mat2[N][M];
3
4   int sum_element(int i, int j)
5   {
6     return mat1[i][j] + mat2[j][i];
7   }
```

在编译这个程序时，GCC 会产生下面这样的汇编代码：

```
1       movl 8(%ebp),%ecx
```

```
2      movl 12(%ebp),%eax
3      leal 0(,%eax,4),%ebx
4      leal 0(,%ecx,8),%edx
5      subl %ecx,%edx
6      addl %ebx,%eax
7      sall $2,%eax
8      movl mat2(%eax,%ecx,4),%eax
9      addl mat1(%ebx,%edx,4),%eax
```

运用你的逆向工程技能，根据这段汇编代码来确定M和N的值。

3.8.5 固定大小的数组

对固定大小的多维数组进行操作的代码，C编译器能够进行多种优化。例如，假设我们将数据类型fix_matrix声明为16×16的整数数组：

```
1   #define N 16
2   typedef int fix_matrix[N][N];
```

图3.24（a）中的代码计算矩阵A和B的乘积的元素i、k。C编译器产生的代码类似于图3.24（b）中所示的那样，这段代码包含很多聪明的优化。编译器认出循环会依次访问数组A的元素A[i][0]，A[i][1]，…，A[i][15]。这些元素占据的是存储器中从数组元素A[i][0]的地址开始的相邻的位置。因此，程序可以用指针变量Aptr来访问这些连续的位置。循环会依次访问数组B的元素B[0][k]，B[1][k]，…，B[15][k]。这些元素占据的是存储器中从数组元素B[0][k]的地址开始的位置，分别相距64个字节。因此，程序可以用指针变量Bptr来访问这些连续的位置。在C中，这个指针会增加16，尽管实际上真实的指针会增加4·16＝64。最后，代码可以用一个简单的计数器来记录需要循环的次数。

code/asm/array.c

```
1   #define N 16
2   typedef int fix_matrix[N][N];
3
4   /* Compute i,k of fixed matrix product */
5   int fix_prod_ele (fix_matrix A, fix_matrix B, int i, int k)
6   {
7       int j;
8       int result = 0;
9
10      for (j = 0; j < N; j++)
11          result += A[i][j] * B[j][k];
12
13      return result;
14  }
```

code/asm/array.c

（a）原始的C代码

code/asm/array.c

```
1   /* Compute i,k of fixed matrix product */
2   int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
```

```
3    {
4       int *Aptr = &A[i][0];
5       int *Bptr = &B[0][k];
6       int cnt = N - 1;
7       int result = 0;
8
9       do {
10          result += (*Aptr) * (*Bptr);
11          Aptr += 1;
12          Bptr += N;
13          cnt--;
14      } while (cnt >= 0);
15
16      return result;
17   }
```

code/asm/array.c

（b）优化过的 C 代码

图 3.24　原始的和优化过的代码，该代码计算固定长度数组的矩阵乘积的元素 i、k

编译器会自动完成这些优化。

我们给出了 fix_prod_ele_opt 的 C 代码，来说明 C 编译器在产生汇编时所使用的优化。下面是这个循环的实际的汇编代码：

Aptr is in %edx, Bptr in %ecx, result in %esi, cnt in %ebx

```
1  .L23:                          loop:
2   movl (%edx),%eax                Compute t = *Aptr
3   imull (%ecx),%eax               Compute v = *Bptr * t
4   addl %eax,%esi                  Add v result
5   addl $64,%ecx                   Add 64 to Bptr
6   addl $4,%edx                    Add 4 to Aptr
7   decl %ebx                       Decrement cnt
8   jns .L23 if >=,                 if >=, goto loop
```

注意，在上面的汇编代码中，所有的指针增加量均乘以伸缩因子值 4。

练习题 3.20

下面的 C 代码将一个固定大小的数组的对角线元素设置为 val：

```
1  /* Set all diagonal elements to val */
2  void fix_set_diag(fix_matrix A, int val)
3  {
4     int i;
5     for (i = 0; i < N; i++)
6       A[i][i] = val;
7  }
```

当编译时，GCC 产生如下汇编代码：

```
1     movl 12(%ebp),%edx
2     movl 8(%ebp),%eax
3     movl $15,%ecx
4     addl $1020,%eax
5     .p2align 4,,7              Added to optimize cache performance
6   .L50:
7     movl %edx,(%eax)
8     addl $-68,%eax
9     decl %ecx
10    jns .L50
```

创建一个 C 代码程序，它使用类似于这段汇编代码中所使用的优化，风格与图 3.24（b）中的代码一致。

3.8.6 动态分配的数组

C 只支持大小在编译时就能知道的多维数组（对于第一维可能有些例外）。在许多应用程序中，我们需要代码能够对动态分配的任意大小的数组进行操作。为此，我们必须显式地写出从多维数组到一维数组的映射。我们可以将数据类型 var_matrix 简单地定义为 int *：

```
typedef int *var_matrix;
```

我们用 Unix 的库函数 calloc 来为一个 $n \times n$ 的整数数组分配和初始化存储：

```
1   var_matrix new_var_matrix(int n)
2   {
3       return (var_matrix) calloc(sizeof(int), n * n);
4   }
```

calloc 函数（ANSI C 文档的一部分[32，40]）有两个参数：每个数组元素的大小和所需数组元素的数目。它试着为整个数组分配空间。如果成功，它会将整个存储器区域初始化为 0，并返回指向第一个字节的指针。如果没有足够的可用空间，它就返回空（null）。

给 C 语言初学者：C、C++和 Java 中的动态存储器分配和释放

在 C 中，堆（一个可以用来存放数据结构的存储器池）中的存储分配是用的库函数 malloc 或 calloc。它们的效果类似于 C++和 Java 中的 new 操作。C 和 C++都要求程序显式地用 free 函数来释放已分配的空间。在 Java 中，释放是由运行时系统通过一个称为 garbage collection（垃圾回收）的进程自动完成的，第 10 章中会讨论这个话题。

然后，我们用行优先顺序的数组下标计算方法确定矩阵元素 i、j 的位置为 $i \cdot n + j$：

```
1   int var_ele(var_matrix A, int i, int j, int n)
2   {
3      return A[(i*n) + j];
4   }
```

翻译成汇编代码是这样的：

```
1    movl 8(%ebp),%edx          Get A
2    movl 12(%ebp),%eax         Get i
3    imull 20(%ebp),%eax        Compute n*i
4    addl 16(%ebp),%eax         Compute n*i + j
5    movl (%edx,%eax,4),%eax    Get A[i*n + j]
```

将这段代码与用来计算固定大小数组的下标的代码相比，我们看到动态版本更加复杂。它必须用一条乘法指令来将 i 增大 n 倍，而不是用一组移位和加法指令。在现代处理器中，这种乘法并不会带来严重的性能损失。

在许多情况中，编译器可以使用相同于我们已描述的固定大小数组的优化原则，来简化大小可变数组的下标计算。例如，图 3.25（a）给出的 C 代码，计算的是两个大小可变矩阵 A 和 B 的乘积的元素 i、k。在图 3.25（b）中，我们给出了一个优化过的版本，它是根据编译原始版本产生的汇编代码逆向生成的。编译器可以利用由循环结构产生的顺序访问模式，消除整数乘法 i*n 和 j*n。在这种情况中，编译器不会产生指针变量 Bptr，而是创建一个我们称为 nTjPk（表示“n 乘以 j 加上 k”）的整数变量，因为相对于原始代码，它的值等于 n*j+k。最开始时，nTjPk 等于 k，每次循环时都增加 n。

code/asm/array.c

```
1   typedef int *var_matrix;
2
3   /* Compute i,k of variable matrix product */
4   int var_prod_ele(var_matrix A, var_matrix B, int i, int k, int n)
5   {
6       int j;
7       int result = 0;
8
9       for (j = 0; j < n; j++)
10          result += A[i*n + j] * B[j*n + k];
11
12      return result;
13  }
```

code/asm/array.c

(a) 原始的 C 代码

code/asm/array.c

```
1   /* Compute i,k of variable matrix product */
2   int var_prod_ele_opt(var_matrix A, var_matrix B, int i, int k, int n)
3   {
4       int *Aptr = &A[i*n];
5       int nTjPk = n;
6       int cnt = n;
7       int result = 0;
8
9       if (n <= 0)
10          return result;
```

```
11
12      do {
13        result += (*Aptr) * B[nTjPk];
14        Aptr += 1;
15        nTjPk += n;
16        cnt--;
17      } while (cnt);
18
19      return result;
20 }
```

code/asm/array.c

(b) 优化过的C代码

图 3.25 计算可变长数组的矩阵乘积的元素i、k的原始和优化过的代码

编译器会自动完成这些优化。

编译器为循环产生代码，其中寄存器%edx保存cnt，%ebx保存Aptr，%ecx保存nTjPk，而%esi保存结果。这段代码如下：

```
1   .L37:                          loop:
2     movl 12(%ebp),%eax             Get B
3     movl (%ebx),%edi               Get *Aptr
4     addl $4,%ebx                   Increment Aptr
5     imull (%eax,%ecx,4),%edi       Multiply by B[nTjPk]
6     addl %edi,%esi                 Add to result
7     addl 24(%ebp),%ecx             Add n to nTjPk
8     decl %edx                      Decrement cnt
9     jnz .L37                       If cnt != 0, goto loop
```

注意，每次循环时，变量B和n都必须从存储器中读出。这是一个寄存器溢出（register spilling）的例子。没有足够的寄存器来保存所有需要的临时数据，因此编译器必须将某些局部变量放在存储器中。此时，编译器会选择溢出变量B和n，因为它们只用读一次——在循环里，它们的值不变。寄存器溢出是IA32一个很常见的问题，因为处理器的寄存器数量太少了。

3.9 异类的数据结构

C提供了两种将不同类型的对象结合到一起来创建数据类型的机制：结构（structure），用关键字struct来声明，将多个对象集合到一个单位中；联合（union），用关键字union来声明，允许用几种不同的类型来引用一个对象。

3.9.1 结构

C的struct声明创建一个数据类型，将可能不同类型的对象聚合到一个对象中。结构的各个组成部分是用名字来引用的。结构的实现类似于数组的实现，因为结构的所有组成部分都存放在存储器中连续的区域内，而指向结构的指针就是结构第一个字节的地址。编译器保存关于每个结构类型

的信息，指示每个域（field）的字节偏移。它以这些偏移作为存储器引用指令中的位移，从而产生对结构元素的引用。

给C语言初学者：将一个对象表示为struct

struct数据类型的构造函数（constructor）是C提供的与C++和Java对象最为接近的东西。它允许程序员保存关于一个数据结构中某些实体的信息，并用名字来引用这些信息。

例如，一个图形程序可能要用结构来表示一个长方形：

```
struct rect {
    int llx;      /* X coordinate of lower-left corner */
    int lly;      /* Y coordinate of lower-left corner */
    int color;    /* Coding of color */
    int width;    /* Width (in pixels) */
    int height;   /* Height (in pixels) */
};
```

我们可以声明一个struct rect类型的变量r，并将它的域值设置为下面这样：

```
    struct rect r;
    r.llx = r.lly = 0;
    r.color = 0xFF00FF;
    r.width = 10;
    r.height = 20;
```

这里表达式r.llx就是结构r的llx域。

将指向结构的指针从一个地方传递到另一个地方，而不是拷贝它们，是很常见的。例如，下面的函数计算长方形的面积，这里，传递给函数的就是一个指向长方形struct的指针：

```
int area(struct rect *rp)
{
    return (*rp).width * (*rp).height;
}
```

表达式(*rp).width间接引用了这个指针，并且选取所得结构的width域。这里必须要用括号，因为编译器会将表达式*rp.width解释为*(rp.width)，而这是非法的。间接引用和域选取的联合使用非常常见，以至于C提供了一种作为替代的标识符->，即rp->width等价于表达式(*rp).width。例如，我们可以写一个函数，它将一个长方形向左旋转90度：

```
void rotate_left(struct rect *rp)
{
    /* Exchange width and height */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width  = t;
}
```

C++和Java的对象比C中的结构要复杂精细得多，因为它们将一组可以被调用以执行计算的方法与一个对象联系起来。在C中，我们可以简单地把这些方法写成普通函数，就像上面所示的函数area和rotate_left。

让我们来看看这样一个例子，考虑下面这样的结构声明：

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

这个结构包括四个域——两个 4 字节 int、一个由三 4 字节 int 组成的数组和一个 4 字节的整数指针——总共是 24 个字节：

偏移	0	4	8			20
内容	i	j	a[0]	a[1]	a[2]	p

注意，数组 a 是嵌入到这个结构中的。上图中顶部的数字给出的是各个域相对结构开始处的字节偏移。

为了访问结构的域，编译器产生的代码要将结构的地址加上适当的偏移。例如，假设 struct rec * 类型的变量 r 放在寄存器%edx 中。然后，下面的代码将元素 r->i 拷贝到元素 r->j:

```
1    movl (%edx),%eax           Get r->i
2    movl %eax,4(%edx)          Store in r->j
```

因为域 i 的偏移量为 0，所以这个域的地址就是 r 的值。为了存储到域 j，代码要将 r 的地址加上偏移量 4。

要产生一个指向结构内部对象的指针，我们只需将结构的地址加上该域的偏移量。例如，我们只用加上偏移量 $8 + 4 \cdot 1 = 12$，就可以得到指针&(r->a[1])。对于在寄存器%eax 中的指针 r 和在寄存器%edx 中的整数变量 i，我们可以用一条指令产生指针&(r->a[i])的值：

```
     r in %eax, i in %edx
1    leal 8(%eax,%edx,4),%ecx   %ecx = &r->a[i]
```

还有最后一个例子，下面的代码实现的是语句：

```
r->p = &r->a[r->i + r->j];
```

开始时 r 在寄存器%edx 中：

```
1    movl 4(%edx),%eax          Get r->j
2    addl (%edx),%eax           Add r->i
3    leal 8(%edx,%eax,4),%eax   Compute &r->[r->i + r->j]
4    movl %eax,20(%edx)         Store in r->p
```

正如这些示例表明的那样，对结构的各个域的选取完全是在编译时处理的。机器代码不包含关于域声明或域名字的信息。

练习题 3.21

考虑下面的结构声明：

```
struct prob {
```

```
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

这个声明说明一个结构可以嵌套在另一个结构中，就像数组可以嵌套在结构中、数组可以嵌套在数组中一样。

下面的过程（省略了某些表达式）是对这个结构进行操作的：

```
void sp_init(struct prob *sp)
{
      sp->s._x    =________;
      sp->p       = ________;
      sp->next    = ________;
}
```

A. 下列域的偏移量是多少（用字节表示）？

```
   p:
 s.x:
 s.y:
next:
```

B. 这个结构总共需要多少字节？

C. 编译器为 sp_init 的主体产生的汇编代码如下：

```
1      movl 8(%ebp),%eax
2      movl 8(%eax),%edx
3      movl %edx,4(%eax)
4      leal 4(%eax),%edx
5      movl %edx,(%eax)
6      movl %eax,12(%eax)
```

根据这些信息，填写出 sp_init 代码中缺失的表达式。

3.9.2 联合

联合提供了一种方式，能够规避C的类型系统，允许以多种类型来引用一个对象。联合声明的语法与结构的语法一样，只不过语义相差比较大。它们不是用不同的域来引用不同的存储器块，而是引用的同一存储器块。

看看下面的声明：

```
struct S3 {
     char c;
     int i[2];
     double v;
```

```
};

union U3 {
    char c;
    int i[2];
    double v;
};
```

域的偏移数据类型 S3 和 U3 的整个大小如下表所示：

类型	c	i	v	大小
S3	0	4	12	20
U3	0	0	0	8

（一会儿我们会看到为什么 S3 中 i 的偏移量为 4，而不是 1。）对于类型 union U3 *的指针 p，p->c、p->i[0]和 p->v 引用的都是数据结构的起始位置。还要注意，一个联合的总的大小等于它最大域的大小。

在一些情况中，联合十分有用。但是，它也引起了一些讨厌的错误，因为它们绕过了 C 类型系统提供的安全措施。一种应用情况是，我们事先知道对一个数据结构中的两个不同域的使用是互斥的，那么将这两个域作为联合的一部分，而不是结构的一部分，会减小分配空间的总量。

例如，假设我们想实现一个二叉树的数据结构，每个叶子节点都有一个 double 的数据值，而每个内部节点都有指向两个孩子节点的指针，但是没有数据。如果我们像这样声明：

```
struct NODE {
    struct NODE *left;
    struct NODE *right;
    double data;
};
```

那么每个节点需要 16 个字节，每种类型的节点都要浪费一半的字节。相反，如果我们这样来声明一个节点：

```
union NODE {
    struct {
        union NODE *left;
        union NODE *right;
    } internal;
    double data;
};
```

那么，每个节点就只需要 8 个字节。如果 n 是一个指针，指向 union NODE *类型的节点，我们用 n->data 来引用叶子节点的数据，而用 n->internal.left 和 n->internal.right 来引用内部节点的孩子。

不过，如果这样编码，就没有办法来确定一个给定的节点到底是叶子节点，还是内部节点。通常的方法是引入一个附加的标志域：

```
struct NODE {
```

```
    int is_leaf;
    union {
        struct {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

这里，对叶子节点来说，域 is_leaf 是 1，而对内部节点来说，该域的值是 0。这个结构总共需要 12 个字节：is_leaf 要 4 个，info.internal.left 和 info.internal.right 各要 4 个，或者 info.data 要 8 个。在这种情况中，相对于给代码造成的麻烦，使用联合带来的好处是很小的。对于有较多域的数据结构，这样的节省会更加吸引人一些。

联合还可以用来访问不同数据类型的位的形式。例如，下面这段代码返回一个 float 作为 unsigned 的位表示：

```
1   unsigned float2bit(float f)
2   {
3       union {
4           float f;
5           unsigned u;
6       } temp;
7       temp.f = f;
8       return temp.u;
9  };
```

在这段代码中，我们以一种数据类型来存储联合中的参数，又以另一种数据类型来访问它。有趣的是，为此过程产生的代码与为下面这个过程产生的代码是一样的：

```
1   unsigned copy(unsigned u)
2   {
3       return u;
4   }
```

这两个过程的主体只有一条指令：

```
1      movl 8(%ebp),%eax
```

这就证明汇编代码中缺乏类型信息。无论参数是一个 float，还是一个 unsigned，它都在相对于 %ebp 偏移量为 8 的地方。过程只是简单地将它的参数拷贝到返回值，不修改任何位。

当用联合来将各种不同大小的数据类型结合到一起时，字节顺序问题就变得很重要了。例如，假设我们写了一个过程，它会以两个 4 字节的 unsigned 的位的形式，创建一个 8 字节的 double：

```
1   double bit2double(unsigned word0, unsigned word1)
2   {
3       union {
4           double d;
5           unsigned u[2];
6       } temp;
```

```
7
8       temp.u[0] = word0;
9       temp.u[1] = word1;
10      return temp.d;
11  }
```

在像IA32这样的小端法（little-endian）机器上，参数word0会是d的低位四个字节，而word1会是高位四个字节。在大端法（big-endian）机器上，这两个参数的角色刚好相反。

练习题 3.22

考虑下面的联合声明：

```
union ele {
   struct {
       int *p;
       int y;
   } e1;
   struct {
       int x;
       union ele *next;
   } e2;
};
```

这个声明说明结构可以嵌套在联合中。

下面的过程（省略了某些表达式）是对一个链表进行操作的，而链表的元素是这些联合：

```
void proc (union ele *up)
{
    up->__________ = *(up->__________) - up->__________;
}
```

A. 下列域的偏移量是多少（用字节表示）？

```
e1.p:
e1.y:
e2.x:
e2.next:
```

B. 这个结构总共需要多少字节？

C. 编译器为proc的主体产生的汇编代码如下：

```
1      movl 8(%ebp),%eax
2      movl 4(%eax),%edx
3      movl (%edx),%ecx
4      movl %ebp,%esp
5      movl (%eax),%eax
6      movl (%ecx),%ecx
7      subl %eax,%ecx
8      movl %ecx,4(%edx)
```

根据这些信息，填写出proc代码中缺失的表达式。**提示**：有些联合引用可以有多种意思的解释。正如你看到的那样，在进行引用的地方，能解决这种歧义。只有一种答案不需要进行任何类型转换，

也不会违反任何类型限制。

3.10 对齐（alignment）

许多计算机系统对基本数据类型的可允许地址做出了一些限制，要求某种类型的对象的地址必须是某个值 k（通常是2、4或8）的倍数。这种对齐限制简化了处理器和存储器系统之间接口的硬件设计。例如，假设一个处理器总是从存储器中取8个字节出来，则地址必须为8的倍数。如果我们能保证所有的double都将它们的地址对齐成8的倍数，那么就可以用一个存储器操作来读或者写值了。否则，我们可能需要执行两次存储器访问，因为对象可能分放在两个8字节存储器块中。

无论数据是否对齐，IA32硬件都能正确工作。不过，Intel还是建议要对齐数据以提高存储器系统的性能。Linux沿用的对齐策略是2字节数据类型（例如short）的地址必须是2的倍数，而较大的数据类型（例如int、int *、float和double）的地址必须是4的倍数。注意，这个要求就意味着一个short类型对象的地址的最低位必须等于0。类似地，任何int类型的对象或指针的地址的最低两位必须都是0。

旁注：Microsoft Windows 的对齐

Microsoft Windows对对齐的要求更严格——任何 k 字节（基本）对象的地址都必须是 k 的倍数。特别地，它要求一个double的地址应该是8的倍数。这种要求提高了存储器性能，代价是浪费了一些空间。Linux中的设计决策可能对i386很好，以前存储器十分缺乏，而存储器总线只有4个字节宽。对于现代处理器来说，Microsoft的对齐策略就是更好的选择了。

命令行选项-malign-double会使Linux上的GCC为double类型的数据使用8字节的对齐，这会提高存储器性能，但是在与用4字节对齐方式下编译的库代码链接时，会导致不兼容。

确保每种数据类型都是按照指定方式来组织和分配的，即每种类型的对象都满足它的对齐限制，就可保证实施对齐。编译器在汇编代码中放入命令，指明全局数据所需的对齐。例如，3.6.6小节中跳转表的汇编代码声明的第2行就包含下面这样的命令（directive）：

```
.align 4
```

这就保证了它后面的数据（在此，是跳转表的开始）会从以4为倍数的地址处开始。因为每个表项长4个字节，后面的元素都会遵守4字节对齐的限制。

分配存储器的库例程（例如malloc）的设计必须使得它们返回的指针能满足最糟糕情况的对齐限制，通常是4或者8。对于有结构的代码，编译器可能需要在域的分配中插入间隙，以保证每个结构元素都满足它的对齐要求，而结构本身对它的起始地址也有一些对齐要求。

比如说，考虑下面的结构声明：

```
struct S1 {
    int i;
    char c;
    int j;
};
```

假设编译器用的是最小的9字节分配，画出图来是这样的：

偏移	0	4	5
内容	i	c	j

它是不可能满足域 i（偏移为 0）和 j（偏移为 5）的 4 字节对齐要求的。所以，编译器在域 c 和 j 之间插入一个 3 字节的间隙（在此用“XXX”表示）：

偏移	0	4	5	8
内容	i	c	XXX	j

结果，j 的偏移量为 8，而整个结构的大小为 12 字节。此外，编译器必须保证任何 struct S1 * 类型的指针 p 都满足 4 字节对齐。用我们前面的符号，让指针 p 的值为 x_p。那么，x_p 必须是 4 的倍数。这就保证了 p->i（地址 x_p）和 p->j（地址 x_p+4）都满足它们的 4 字节对齐要求。

另外，编译器可能需要添加一些填充到结构的末尾，这样结构数组的每个元素都会满足它的对齐要求。例如，看看下面这个结构声明：

```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

如果我们将这个结构打包成 9 个字节，只要保证结构的起始地址满足 4 字节对齐要求，我们仍然能够保证满足域 i 和 j 的对齐要求。不过，考虑下面的声明：

```
struct S2 d[4];
```

分配 9 个字节，是不可能满足 d 的每个元素的对齐要求的，这是因为这些元素的地址分别为 x_d、x_d+9、x_d+18 和 x_d+27。

编译器会为结构 S1 分配 12 个字节，最后 3 个字节是浪费的空间：

偏移	0	4	8	9
内容	i	j	c	XXX

这样一来，d 的元素的地址分别为 x_d、x_d+12、x_d+24 和 x_d+36。只要 x_d 是 4 的倍数，所有的对齐限制就都可以满足了。

练习题 3.23

对下面每个结构声明，确定每个域的偏移量、结构总的大小以及在 Linux/IA32 下它的对齐要求。

```
A. struct P1 { int i; char c; int j; char d; };
B. struct P2 { int i; char c; char d; int j; };
C. struct P3 { short w[3]; char c[3] };
D. struct P4 { short w[3]; char *c[3] };
E. struct P3 { struct P1 a[2]; struct P2 *p };
```

3.11 综合：理解指针

指针是 C 语言的一个重要特色。它们提供一种统一方式，能够远程访问数据结构。对于编程新

下来说，指针总是会带来很多的困惑，但是基本的概念其实非常简单。图 3.26 中的代码说明了许多这样的概念。

```
1   struct str {    /* Example Structure */
2       int t;
3       char v;
4   };
5
6   union uni {  /* Example Union */
7       int t;
8       char v;
9   } u;
10
11  int g = 15;
12
13  void fun(int* xp)
14  {
15      void (*f)(int*) = fun; /* f is a function pointer */
16
17      /* Allocate structure on stack */
18      struct str s = {1,'a'}; /* Initialize structure */
19
20      /* Allocate union from heap */
21      union uni *up = (union uni *) malloc(sizeof(union uni));
22
23      /* Locally declared array */
24      int *ip[2] = {xp, &g};
25
26      up->v = s.v+1;
27
28      printf("ip = %p, *ip = %p, **ip = %d\n",
29             ip, *ip, **ip);
30      printf("ip+1 = %p, ip[1] = %p, *ip[1] = %d\n",
31             ip+1, ip[1], *ip[1]);
32      printf("&s.v = %p, s.v = '%c'\n", &s.v, s.v);
33             printf("&up->v = %p, up->v = '%c'\n", &up->v, up->v);
34      printf("f = %p\n", f);
35      if (--(*xp) > 0)
36      f(xp);                        /* Recursive call of fun */
37  }
38
39  int test()
40  {
41      int x = 2;
42      fun(&x);
43      return x;
44  }
```

图 3.26 用来说明 C 中指针使用的代码

在 C 中，指针可以指向任何数据类型。

- 每个指针都有一个类型。这个类型表明指针指向的对象是哪一类的。在我们的示例代码中，我们看到了下面这样一些指针类型：

指针类型	对象类型	指　针
int *	int	xp, ip[0], ip[1]
union uni *	union uni	up

注意，在前面那张表中，我们既指出了指针本身的类型，也指出了它所指向的对象的类型。通常，如果对象类型为 *T*，那么指针的类型为 **T*。特殊的 void *类型代表通用指针。比如说，malloc 函数返回一个通用指针，然后它再被强制类型转换成一个有类型的指针（第 21 行）。

- 每个指针都有一个值。这个值是某个指定类型的对象的地址。特殊的 NULL（0）值表示该指针没有指向任何地方。待会儿，我们会看看我们的指针的值。
- 指针是用&运算符创建的。这个运算符可以应用到任何 lvalue 类的 C 表达式上，也就是可以出现在赋值语句左边的表达式，这样的例子包括变量以及结构、联合和数组的元素。在我们的示例代码中，我们看到这个操作符应用到全局变量 g 上（第 24 行），应用到结构元素 s.v 上（第 32 行），应用到联合元素 up->v 上（第 33 行），以及应用到局部变量 x 上（第 42 行）。
- *操作符用于指针的间接引用。其结果是一个值，它的类型与该指针的类型相关。我们看到间接引用应用到 ip 和*ip 上（第 29 行），应用到 ip[1]上（第 31 行），以及应用到 xp 上（第 35 行）。此外，表达式 up->v（第 33 行）既间接引用了指针 up，同时还选取了域 v。
- 数组与指针是紧密联系的。可以引用一个数组的名字（但是不能修改），就好像它是一个指针变量一样。数组引用（例如，a[3]）与指针运算和间接引用（例如，*(a+3)）有一样的效果。我们可以在第 29 行看到这一点，我们打印出数组 ip 的指针值，并用*ip 引用它的第一项（元素 0）。
- 指针也可以指向函数。这提供了一个很强大的存储（storing）和传递代码引用的功能，这些代码可以被程序的某个其他部分调用。看看变量 f（第 15 行），它被声明为一个指向函数的变量，该函数以一个 int *作为参数，并返回 void。赋值语句使 f 指向 fun。当在后面我们使用 f（第 36 行）时，我们是在进行递归调用。

给 C 语言初学者：函数指针

函数指针声明的语法对程序员新手来说是特别难以理解的。对于这样一个声明：

```
void (*f)(int*);
```

要从里（从“f”开始）往外读。因此，我们看到像“(*f)”表明的那样，f 是一个指针。像“(*f)(int*)”表明的那样，它是一个指针，指向一个以一个 int *作为参数的函数。最后，我们看到，它是一个指向一个以 int *作为参数并返回 void 的函数的指针。

***f 两边的括号是必须的，因为否则声明**

```
void *f(int*);
```

就要读成

```
(void *) f(int*);
```

也就是，它会被解释成一个函数原型，声明了一个函数 f，它以一个 int *作为参数并返回一个 void *。

Kernighan 和 Ritchie [40，5.12 节]给出了一个有关阅读 C 声明的很有帮助的教程。

我们的代码包含很多对 printf 的调用，打印出一些指针（用指令%p）和值。在执行时，产生下面这样的输出：

```
1   ip       = 0xbfffefa8, *ip = 0xbfffefe4, **ip = 2  ip[0] = xp. *xp = x = 2
2   ip+1 = 0xbfffefac, ip[1] = 0x804965c, *ip[1] = 15  ip[1] = &g. g = 15
3   &s.v = 0xbfffefb4, s.v = 'a'                        s in stack frame
4   &up->v   = 0x8049760, up->v = 'b'                   up points to area in heap
5   f        = 0x8048414                                f points to code for fun
6   ip       = 0xbfffef68, *ip = 0xbfffefe4, **ip = 1  ip in new frame, x = 1
7   ip+1 = 0xbfffef6c, ip[1] = 0x804965c, *ip[1] = 15  ip[1] same as before
8   &s.v = 0xbfffef74, s.v = 'a'                        s in new frame
9   &up->v   = 0x8049770, up->v = 'b'                   up points to new area in heap
10  f        = 0x8048414                                f points to code for fun
```

我们看到，这个函数执行了两次——第一次是从 test 中直接调用（第 42 行），而第二次是间接的递归调用（第 36 行）。我们可以看出，打印出来的指针值都对应于地址。那些从 0xbfffef 开始的指针指向栈中的位置，而其他的是全局存储的一部分（0x804965c），或是可执行代码的一部分（0x8048414），或者是堆中的位置（0x8049760 和 0x8049770）。

数组 ip 被初始化了两次——每次调用 fun 都初始化一次。第二次的值（0xbfffef68）小于第一次的值（0xbfffefa8），这是因为栈是向下增长的。不过，数组的内容两次都是一样的。数组元素 0（*ip）是一个指向 test 栈帧中变量 x 的指针，元素 1 是一个指向全局变量 g 的指针。

我们可以看到结构 s 也被初始化了两次，两次都是在栈中，而变量 up 指向的联合是在堆中分配的。

最后，变量 f 是一个指向函数 fun 的指针。在反汇编代码中，我们看到如下 fun 的初始化代码：

```
1     08048414 <fun>:
2     8048414:  55                    push    %ebp
3     8048415:  89 e5                 mov     %esp,%ebp
4     8048417:  83 ec 1c              sub     $0x1c,%esp
5     804841a:  57                    push    %edi
```

打印出来的指针 f 的值 0x8048414 就是 fun 的代码中第一条指令的地址。

给 C 语言初学者：向函数传递参数

其他语言（例如 Pascal）提供两种方式来向过程传递参数——传值（by value）和引用（by reference）。传值是指调用者提供实际的参数值，而引用是指调用者提供一个指向该值的指针。在 C 中，所有的参数都是传值的，但是我们可以通过显式地产生一个指向一个值的指针，并把该指针传递给过程，从而实现了引用参数的效果。函数 `fun(&x)`（图 3.26）中的参数 xp 就是这样的。第一次调用 `fun(&x)` 时（第 42 行），给了函数一个对 test 中局部变量 x 的引用。每次调用 fun 时，这个变量都会减小，从而在两次调用之后，递归会停止。

3.12 现实生活：使用 GDB 调试器

GNU 的调试器 GDB 提供了许多有用的特性来支持对机器级程序的运行时评估和分析。我们试图用本书中的示例和练习，通过阅读代码，来推断出程序的行为。有了 GDB，通过观察正在运行的程序，同时又对程序的执行有相当的控制，这就使得研究程序的行为变为可能。

图3.27给出了一些GDB命令的例子，在使用机器级IA32程序时，会有所帮助。先运行OBJDUMP来获得程序的反汇编版本，是很有好处的。我们的示例都是基于对文件 prog 运行 GDB 的，程序的描述和反汇编都在第 110 页。我们用下面的命令行来启动 GDB：

```
unix> gdb prog
```

通常的方法是在程序中感兴趣的地方附近设置断点。断点可以设置在函数入口后面，或是设置在一个程序的地址处。在程序执行过程中，遇到一个断点时，程序会停下来，并将控制返回给用户。在断点处，我们能够以各种方式查看各个寄存器和存储器位置。我们也可以单步跟踪程序，一次只执行几条指令，或是前进到下一个断点。

命令	效果
开始和停止	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
断点	
`break sum`	Set breakpoint at entry to function sum
`break *0x80483c3`	Set breakpoint at address 0x80483c3
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
执行	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like stepi, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
检查代码	
`disas`	Disassemble current function
`disas sum`	Disassemble function sum
`disas 0x80483b7`	Disassemble function around address 0x80483b7
`disas 0x80483b7 0x80483c7`	Disassemble code within speci.ed address range
`print /x $eip`	Print program counter in hex
检查数据	
`print $eax`	Print contents of %eax in decimal
`print /x $eax`	Print contents of %eax in hex
`print /t $eax`	Print contents of %eax in binary
`print 0x100`	Print decimal representation of 0x100
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of %ebp plus 8 in hex

*print *(int *) 0xbffff890*	Print integer at address 0xbffff890
*print *(int *) ($ebp+8)*	Print integer at address %ebp + 8
x/2w 0xbffff890	Examine two (4-byte) words starting at address 0xb-ffff890
x/20b sum	Examine .rst 20 bytes of function sum
有用的信息	
info frame	Information about current stack frame
info registers	Values of all the registers
help	Get information about GDB

图 3.27 GDB 命令示例

这些例子说明了几种 GDB 支持机器级程序调试的方式。

正如我们的示例表明的那样，GDB 的命令语法有点含混晦涩，但是在线帮助信息（用 GDB 的 help 命令调用）能克服这些毛病。

3.13 存储器的越界引用和缓冲区溢出

我们已经看到，C 对于数组引用不进行任何边界检查，而且局部变量和状态信息（例如寄存器值和返回指针）都存放在栈中。这两种情况结合到一起就能导致严重的程序错误，一个对越界的数组元素的写操作破坏了存储在栈中的状态信息。然后，当程序使用这个被破坏的状态，试图重新加载寄存器或执行 ret 指令时，就会出现很严重的错误。

一种特别常见的状态破坏称为缓冲区溢出（buffer overflow）。通常，在栈中分配某个字节数组来保存一个字符串，但是字符串的长度超出了为数组分配的空间。下面这个程序示例就说明了这个问题：

```
1   /* Implementation of library function gets() */
2   char *gets(char *s)
3   {
4       int c;
5       char *dest = s;
6       while ((c = getchar()) != '\n' && c != EOF)
7       *dest++ = c;
8       *dest++ = '\0'; /* Terminate String */
9       if (c == EOF)
10          return NULL;
11      return s;
12  }
13
14  /* Read input line and write it back */
15  void echo()
16  {
17      char buf[4]; /* Way too small! */
18      gets(buf);
19      puts(buf);
```

```
20  }
```

前面的代码给出了一个库函数 gets 的实现，用来说明这个函数的严重问题。它从标准输入读入一行，在遇到一个“\n”字符或某个错误情况时停止。它将这个字符串拷贝到参数 s 指明的位置，并在字符串结尾加上 null 字符。在函数 echo 中，我们使用了 gets，这个函数只是简单地从标准输入中读入，再回送到标准输出。

gets 的问题是它没有办法确定是否为保存整个字符串分配了足够的空间。在我们的 echo 示例中，我们故意将缓冲区设得非常小——只有四字节长。任何长度超过 3 个字符的字符串都会导致写越界。

研究 echo 汇编代码的这一部分，看看栈是如何组织的：

```
1  echo:
2    pushl %ebp                Save %ebp on stack
3    movl %esp,%ebp
4    subl $20,%esp             Allocate space on stack
5    pushl %ebx Save %ebx
6    addl $-12,%esp            Allocate more space on stack
7    leal -4(%ebp),%ebx        Compute buf as %ebp-4
8    pushl %ebx                Push buf on stack
9    call gets Call gets
```

在这个例子中，我们可以看到，程序总共为局部存储（storage）分配了 32 个字节（第 4 行和第 6 行）。不过，字符数组 buf 的位置在%ebp 下方四个字节处（第 7 行）。图 3.28 给出了得到的栈结构。正如看到的那样，所有对 buf[4]～buf[7]的写都会导致%ebp 的保存值被破坏。当程序随后试图以它为栈指针进行恢复时，所有后来的栈引用都会是非法的。所有对 buf[8]～buf[11]的写都会导致返回地址被破坏。当在函数结尾执行 ret 指令时，程序会“返回”到错误的地址。像这个示例说明的那样，缓冲区溢出可能导致程序出现严重的错误。

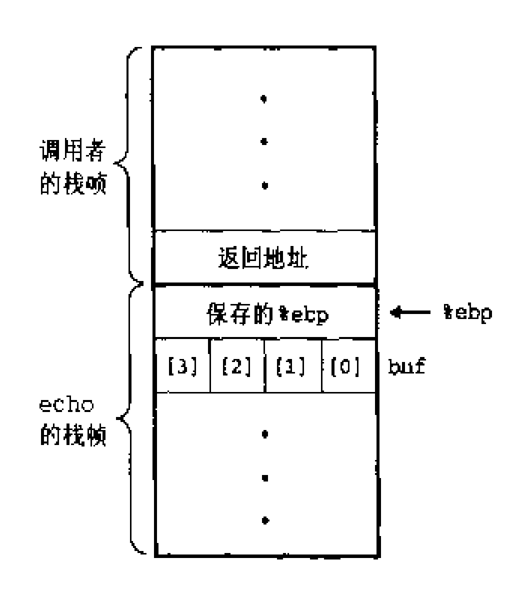


图 3.28 echo 函数的栈组织

字符数组 buf 就在保存的状态下面，对 buf 的写越界会破坏程序的状态。

我们的 echo 代码很简单，但是有点太随意了。更好一点的版本是使用 fgets 函数，它包括一个

一个参数，限制待读入的最大字节数。家庭作业3.37要求你写出一个能处理任意长度输入字符串的echo函数。通常，使用gets或其他能导致存储溢出的函数，都是不好的编程习惯。当编译一个含有调用gets的文件时，C编译器甚至会产生这样的出错信息："the gets function is dangerous and should not be used（gets函数很危险，不应使用）。"

code/asm/bufovf.c

```
1   /* This is very low quality code.
2      It is intended to illustrate bad programming practices.
3      See Practice Problem 3.24. */
4   char *getline()
5   {
6       char buf[8];
7       char *result;
8       gets(buf);
9       result = malloc(strlen(buf));
10      strcpy(result, buf);
11      return(result);
12  }
```

code/asm/bufovf.c

C代码

```
1   08048524 <getline>:
2    8048524:  55                  push %ebp
3    8048525:  89 e5               mov %esp,%ebp
4    8048527:  83 ec 10            sub $0x10,%esp
5    804852a:  56                  push %esi
6    804852b:  53                  push %ebx
     Diagram stack at this point
7    804852c:  83 c4 f4            add $0xfffffff4,%esp
8    804852f:  8d 5d f8            lea 0xfffffff8(%ebp),%ebx
9    8048532:  53                  push %ebx
10   8048533:  e8 74 fe ff ff      call 80483ac <_init+0x50>   gets
     Modify diagram to show values at this point
```

对gets调用的反汇编

图3.29 练习题3.24的C和反汇编代码

练习题3.24

图3.29给出了一个函数的（不太好的）实现，这个函数从标准输入读入一行，将字符串拷贝到新分配的存储，并返回一个指向结果的指针。

考虑下面这样的场景：过程getline被调用，返回地址等于0x8048643，寄存器%ebp等于0xbffffc94，寄存器%esi等于0x1，而寄存器%ebx等于0x2。输入字符串为"012345678901"，程序会因为段错误（segmentation fault）而中止。运行GDB，确定出错误是在执行getline的ret指令时发生的。

A. 填写下图，说出尽可能多的关于在执行完反汇编代码中第6行指令后栈的信息。在右边标注出存储在栈中的数字的意思（例如，"返回地址"），在方框中写出它们的十六进制值。每个方框都代表4个字节。另外，还需指出%ebp的位置。

08 04 86 43	返回地址

B. 修改你的图，以展现调用 gets 的影响（第 10 行）。

C. 程序应该试图返回到什么地址？

D. 当 getline 返回时，哪个（些）寄存器被破坏了？

E. 除了可能会缓冲区溢出以外，getline 的代码还有哪两个错误？

缓冲区溢出的一个更加致命的使用就是让程序执行它本来不愿意执行的函数。这是一种最常见的通过计算机网络攻击系统安全的方法。通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为 exploit code，另外，还有一些字节会用一个指向缓冲区中那些可执行代码的指针覆盖掉返回指针。所以，执行 ret 指令的效果就是跳转到 exploit code。

在一种攻击形式中，exploit code 会使用系统调用启动一个 shell 程序，提供给攻击者一组操作系统的函数。在另一种攻击形式中，exploit code 执行一些未授权的任务，修复对栈的破坏，然后第二次执行 ret 指令，（看上去好像）正常返回给调用者。

让我们来看一个例子，著名的 Internet 蠕虫病毒在 1988 年 11 月通过 Internet 以四种不同的方法获取对许多计算机的访问。一种是对 finger 守护进程 fingerd 的缓冲区溢出攻击，fingerd 是通过 FINGER 命令来服务请求的。通过以一个适当的字符串调用 FINGER，蠕虫可以使远程的守护进程缓冲区溢出并执行一段代码，该代码能让蠕虫访问远程系统。一旦蠕虫获得了对系统的访问，它就能自我复制，几乎完全地消耗掉机器上所有的计算资源。因此，在安全专家抓住如何消除这种蠕虫的方法之前，成百上千的机器实际上都瘫痪了。这种蠕虫的始作俑者最后被抓住并被起诉。他被判处三年徒刑（缓期执行）、400 个小时的社区服务以及 10500 美元的罚款。不过，即使到今天，人们还是在不断地发现使他们容易遭受缓冲区溢出攻击的系统安全漏洞，这更加突显了小心仔细编写程序的必要性。任何到外部环境的接口都应该是“防弹的”，这样，外部 agent 的行为才不会导致系统出现错误。

旁注：蠕虫和病毒

蠕虫和病毒都是试图在计算机中传播它们自己的代码。正如 Spafford[73]讲述的那样，蠕虫（worm）是这样一种程序，它可以自己运行，并且能够将一个完全有效的自己传播到其他机器。与此相应地，病毒（virus）是这样一段代码，它能将自己添加到包括操作系统在内的其他程序中，但它不能独立运行。在一些大众媒体中，术语“病毒”用来指各种不同的在系统间传播攻击代码的策略，所以你可能会听到人们把本来应该叫做“蠕虫”的东西称为了“病毒”。

在家庭作业 3.38 中，你可以获得准备缓冲区溢出攻击的第一手经验。注意，我们不能原谅任何

用这种或其他任何方法来获得对系统的未被授权的访问。未经许可闯入计算机系统与闯入一幢建筑是一样的——是一种犯罪行为，即使犯罪者并没有恶意。我们给出这样一个作业有两个原因：首先，它要求对机器语言编程有很深的了解，将许多问题结合了起来，例如栈的组织、字节排序以及指令编码。其次，通过讲解缓冲区溢出攻击是如何进行的，我们希望你能了解到编写不允许这种攻击的代码的重要性。

旁注：借助于缓冲区溢出与 Microsoft 作战

在 1999 年 7 月，Microsoft 提出了一种即时消息（IM）系统，其客户端与流行的美国在线（AOL）的 IM 服务器兼容。这就使得 Microsoft 的 IM 用户可以和 AOL 的 IM 用户聊天。不过，一个月后，Microsoft 的 IM 用户突然神秘地不能与 AOL 的 IM 用户聊天了。Microsoft 发布了更新的客户端，恢复了对 AOL IM 系统的服务，但是几天之内，这些客户端也又不能工作了。尽管 Microsoft 版本的客户端不断地尝试模仿 AOL IM 的协议，不知怎么地，AOL 就是能够确定一个用户是否运行的是 AOL 版本的 IM 客户端。

AOL 客户端代码容易遭受缓冲区溢出攻击。这很可能是 AOL 代码中一个因疏忽所致的"特色"。AOL 利用它自己代码中的这个错误，通过在用户登陆时攻击客户端，来发现假冒者。AOL 的 exploit code 从客户端的存储器映像中取出很少量的位置样本，将它们打成一个网络包，发送回服务器。如果服务器没有收到这样的包，或者如果收到的包与预期的 AOL 客户端的"足迹"不匹配，那么服务器就会假定这个客户端不是 AOL 的客户端，并拒绝它的访问。所以，如果其他 IM 客户端，例如 Microsoft 的客户端，想访问 AOL 的 IM 服务器，他们不仅要加入 AOL 客户端中存在的缓冲区溢出错误，而且在适当的存储器位置中，还要有完全相同的二进制代码和数据。但是，一旦他们使这些位置相匹配了，将他们新的客户端程序向用户分发了，AOL 只需简单地修改它的 exploit code，取出客户端存储器映像中不同的位置样本。很明显，这是一场非 AOL 客户端永远也不可能赢的战争！

整个事件是一波三折的。关于客户端错误和 AOL 利用这个错误的消息最早泄露出来，是有人冒充名为 Phil Bucking 的独立咨询顾问，向有名的安全专家 Richard Smith 发了一封电子邮件，讲述了这个消息。Smith 进行了一些跟踪，发现这封邮件实际上是从 Microsoft 内部发出的，后来 Microsoft 承认它的一个雇员发了这封邮件[51]。而在这场论战的另一方，AOL 既不承认有这样一个错误，也不承认他们利用这个错误，即使是在澳大利亚的 Geoff Chapell 将结论性的证据公之于众之后。

那么，在这个事件中，谁违反了哪些行为规范呢？首先，AOL 没有义务向非 AOL 客户端开发它的 IM 系统，所以他们阻止 Microsoft 是正当的。另一方面，使用缓冲区溢出是件很棘手的事情。一个很小的错误可能就会导致客户端计算机崩溃，而且它使得系统更容易遭受外部主体的攻击（虽然没有证据显示已经发生了这样的事情）。Microsoft 将 AOL 故意使用缓冲区溢出公之于众是对的。不过，用 Phil Bucking 这样的伎俩来散布这个消息，无论是从道德上来说，还是从公共关系的角度来看，明显都是错误的。

3.14 *浮点代码

处理浮点值的指令集是 IA32 体系结构最不优美的特性之一。在最早的 Intel 机器中，浮点是由一个独立的协处理器来完成的，这个部件有它自己的寄存器和处理能力，能够执行一部分指令。这

个协处理器是由名为 8087、80287 和 i387 的独立芯片实现的，且伴随着处理器芯片 8086、80286 和 i386。在这些产品的开发过程中，芯片的容量已经不足以在一块芯片上既包括主处理器又包括浮点协处理器的。另外，廉价的机器会省去浮点硬件，只用软件来完成浮点操作（非常慢!）。从 i486 开始，浮点就作为 IA32 CPU 芯片的一部分了。

1980 年，最早的 8087 协处理器的问世赢得了很高的赞誉。它是第一个单芯片浮点单元（FPU），同时也是 IEEE 浮点的第一个实现。作为协处理器运行时，在主处理器取出浮点指令后，FPU 会接过它们完成执行。FPU 和主处理器之间有最少限度的连接。将数据从一个处理器传递到另一个，需要发送方处理器写存储器，接收方处理器再从存储器中读取。直到今天 IA32 浮点指令集中还保留有这些设计的遗迹。另外，1980 年的编译技术比今天的简陋得多。对于优化编译器来说，IA32 浮点的许多特性都是很难的目标。

3.14.1 浮点寄存器

浮点单元包括 8 个浮点寄存器，但是和普通寄存器不一样，这些寄存器是被当成一个浅栈（shallow stack）来对待的。这些寄存器分别标识为%st(0)、%st(1)，等等，直到%st(7)。其中，%st(0) 在栈顶。当压入栈中的值超过 8 个时，栈底的那些值就会消失。

大多数算术指令不会直接引用寄存器，而是从栈中弹出它们的源操作数，计算结果，再将结果压入栈中。在 20 世纪 70 年代，栈结构还被认为是很聪明的想法，因为它们提供了一种简单的对算术指令求值的机制，同时它们也允许指令的密集编码（dense coding）。随着编译技术的进步，同时，指令编码所需要的存储器也不再是很关键的资源，这些属性就不再重要了。写编译器的人会更高兴有一组更大的、使用方便的浮点寄存器。

旁注：其他基于栈的语言

基于栈的解释器仍然被广泛用做高级语言和它到实际机器上的映射之间的中间表示。其他基于栈的求值程序的示例包括 Java 字节代码、Java 编译器产生的中间格式，以及 PostScript 页面格式化语言。

将浮点寄存器组织成一个有界的栈，使得编译器很难用这些寄存器来存放一个调用其他过程的过程的局部变量。对于局部变量的存放，我们已经看到，有些通用寄存器可以被指定为由被调用者保存，因此，可以用来保存跨过程调用的局部变量。这种指定对 IA32 浮点寄存器来说是不可能的，因为它的标识随着值压入栈中和从栈中弹出是变化的。一个压栈操作会使%st(0)中的值现在在%st(1)中。

另一方面，它会将浮点寄存器作为真正的栈来对待，每次过程调用时，都将本地值压入其中。不幸的是，很快就会导致栈溢出，因为只有够放 8 个值的位置。作为代替，编译器产生的代码会在调用另一个过程之前，将每个本地浮点值都压入到主程序栈中，然后在返回时把它们取出来。这样引起的存储器访问操作会降低程序的性能。

像 2.4.6 节中说明的那样，IA32 浮点寄存器的宽都是 80 位。它们以家庭作业 2.58 中描述的扩展精度格式来对数字编码。当从存储器加载到浮点寄存器时，所有的单精度和双精度数都转换成这种格式。运算总是以扩展精度格式执行的。当存回存储器中时，数字会从扩展精度转换成单精度或双精度格式。

3.14.2 栈的表达式求值

为了理解 IA32 是如何用它的浮点寄存器作为栈的，让我们来看看基于栈来求值的一个更加抽

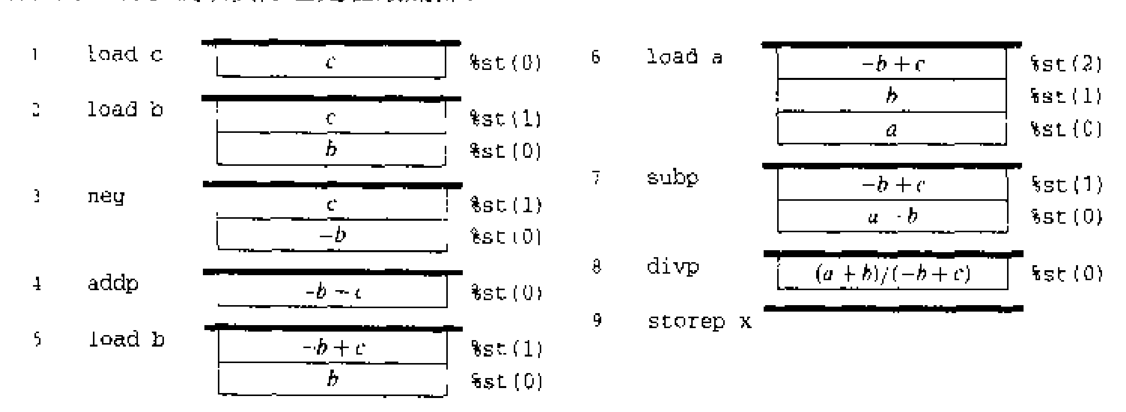
象的版本。假设我们有一个算术单元，它用栈来保存中间结果，其指令集如图 3.30 所示。比如说，所谓的 RPN（Reverse Polish Notation，逆波兰表示）袖珍计算器就提供了这种特性。除了这个栈，该单元还有一个可以保存值的存储器，我们用名字来引用这些值，例如 a、b 和 x。如图 3.30 表明的，我们能够用 load 指令将存储器值压入这个栈中。storep 操作弹出栈顶元素，并将结果存放到存储器中。单操作数的操作，例如 neg（求反），将栈顶元素作为它的参数，并用结果覆盖这个元素。双操作数的操作，例如 addp 和 multp，以栈顶两个元素作为参数。它们会将两个参数都弹出，然后将结果压回栈中。我们在存储、加法、减法、乘法和除法指令后面加上后缀“p”，是为了强调这些指令弹出了它们的操作数。

指 令	效 果
load *S*	将 *S* 处的值压入栈中
storep *D*	弹出栈顶元素并存储在 *D* 处
neg	栈顶元素取负
addp	弹出两个栈顶元素；压入它们的和
subp	弹出两个栈顶元素；压入它们的差
multp	弹出两个栈顶元素；压入它们的积
divp	弹出两个栈顶元素；压入它们的比值

图 3.30 假设的栈指令集

这些指令用来说明基于栈的表达式求值。

作为一个示例，考虑表达式 x=(a-b) / (-b+c)。我们可以将这个表达式翻译成下面的代码。在每一行代码旁边，都给出了浮点寄存器栈的内容。为了与我们前面的惯例保持一致，我们画的栈是向下增长的，所以栈顶实际上是在最底部。



就像这个例子说明的那样，将一个算术表达式转换成栈代码是一个天然的递归过程。我们的表达式记法规定四种类型的表达式，具有下列翻译规则：

1. 格式为 Var 的变量引用。是用指令 load Var 来实现的。

2. 格式为-Expr 的单操作数操作。这是用先产生 Expr 的代码，然后再跟一条 neg 指令来实现的。

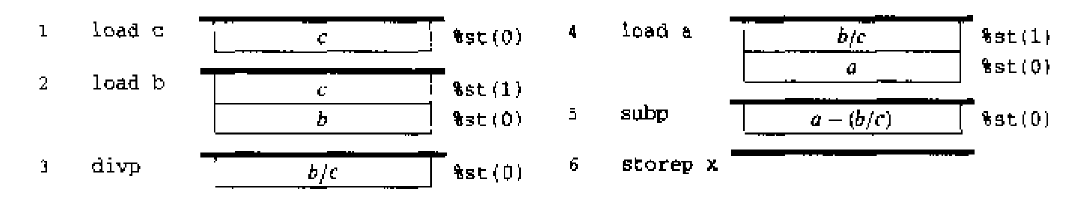
3. 格式为 $Expr_1 + Expr_2$、$Expr_1 - Expr_2$、$Expr_1 * Expr_2$ 或 $Expr_1 / Expr_2$ 的双操作数操作。它的实现是产生 $Expr_2$ 的代码，然后是 $Expr_1$ 的代码，然后是一条 addp、subp、multp 或 divp 指令。

4. 格式为 Var = Expr 的赋值操作。这是通过先产生 Expr 的代码，然后跟一条 storep Var 指令来实现的。

作为一个示例，看看这样一个表达式 x = a - b / c。因为除法的优先级高于减法，这个表达式加上括号就变成了 x = a - (b / c)。因此递归过程会像这样进行：

1. 产生 Expr ≐ a - (b/c) 的代码：
   (a) 产生 $Expr_2$ ≐ b/c 的代码：
      i. 用指令 load c 产生 $Expr_2$ ≐ c 的代码。
      ii. 用指令 load b 产生 $Expr_1$ ≐ b 的代码。
      iii. 产生指令 divp。
   (b) 用指令 load a 产生 $Expr_1$ ≐ a 的代码。
   (c) 产生指令 subp。
2. 产生指令 storep x。

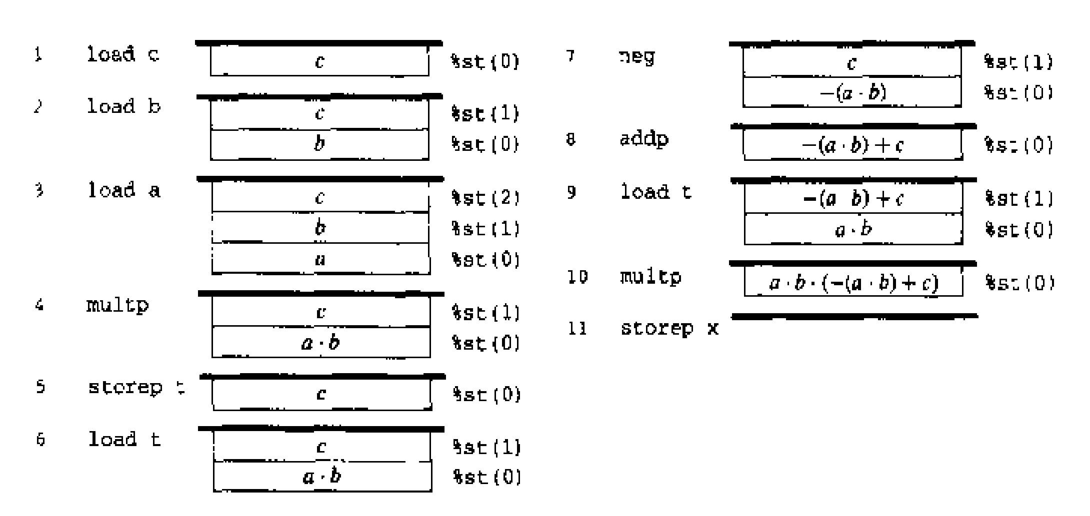
整体效果就是产生下面这样的栈代码：



练习题 3.25

产生表达式 x = a*b/c * - (a+b*c)的栈代码。画出每一步代码的栈内容。记住要遵守 C 的有关优先级和结合性规则。

当我们想多次使用某些计算结果时，栈求值就变得更加复杂了。例如，考虑这样的表达式 x = (a*b)*(-(a*b)+c)。为了效率，我们想只计算 a*b 一次，但是我们的栈指令不提供一种方式将值保存在栈中，一旦这个值被用过。因此，使用图 3.30 中列出的这样一组指令，我们会需要将中间结果 a*b 存储在存储器中某个位置，比如说 t，每次要使用时就取出这个值。得到下面这样的代码：



这种方法的缺点就是增加了额外的存储器访问操作，即使是在寄存器栈有足够的容量存放中间结果时。IA32 浮点单元避免了这种低效率，引入了算术指令的变种，将它们的第二个操作数留在栈中，可以用任意栈值作为它们的第二个操作数。另外，它还提供一条指令，可以将栈顶元素与任何其他元素进行交换。虽然这些扩展可以用来产生更有效的代码，但是将算术表达式翻译成栈代码的

简单而优美的算法丢失了。

3.14.3 浮点数据的传送和转换操作

用记符%st(i)来引用浮点寄存器，这里 i 代表相对于栈顶的位置。值 i 的范围为 0~7。寄存器%st(0)是栈顶元素，%st(1)是第二个，依此类推。也可以用%st 来引用栈顶元素。当一个新值压入栈中时，寄存器%st(7)中的值就丢失了。当从栈中弹出时，%st(7)中的新值是不可预测的。编译器产生的代码必须能在寄存器栈有限的容量中工作。

图 3.31 给出的指令集是用来将值压入浮点寄存器栈中的。第一组指令从存储器位置中读，这里参数 Addr 是存储器地址，它按照图 3.3 中列出的某种存储器操作数格式给出。这些指令是以假定的源操作数的格式来区分的，因此必须从存储器中读出一组字节。回忆一下符号 M_b[Addr]，表示对起始地址为 Addr 的 b 个字节的访问。在将操作数压入栈中之前，所有这些指令都会将它转换成扩展精度格式。最后的加载指令 fld 用来复制一个栈的值。也就是，它将浮点寄存器%st(i)的一个副本压入栈中。例如，指令 fld %st(0)将栈顶元素的一个副本压入栈中。

指令		源格式	源位置
flds	*Addr*	单精度	M_4[*Addr*]
fldl	*Addr*	双精度	M_8[*Addr*]
fldt	*Addr*	扩展精度	M_{10}[*Addr*]
fildl	*Addr*	整数	M_4[*Addr*]
fld	%st(i)	扩展精度	%st(i)

图 3.31 浮点加载指令

所有的指令将操作数转换成扩展精度格式，然后压入寄存器栈中。

图 3.32 给出了将栈顶元素存储在存储器或另一个浮点寄存器中的指令。“弹出”有两个版本，一种是将栈顶元素弹出栈（类似于我们假设的栈求值器中的 storep 指令），一种是非弹出版本，将源值留在栈顶上。同浮点加载指令一样，指令的不同变种产生的结果格式也不同，因而会存储不同数目的字节。第一组指令是将结果存到存储器中。地址是用图 3.3 中列出的存储器操作数格式中的某一种指定的。第二组指令是将栈顶元素拷贝到另外一个浮点寄存器中。

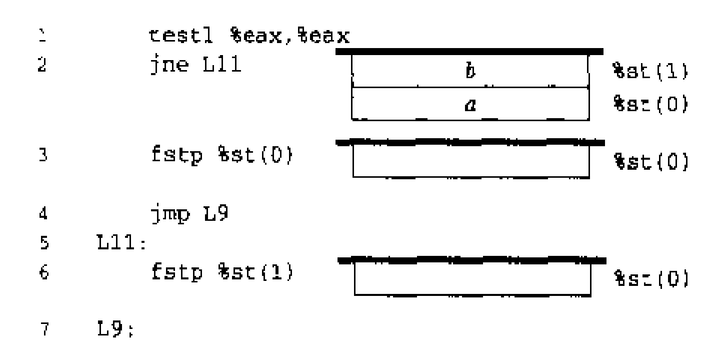
指令		弹出（Y/N）	目标格式	目标位置
fsts	*Addr*	N	单精度	M_4[*Addr*]
fstps	*Addr*	Y	单精度	M_4[*Addr*]
fstl	*Addr*	N	双精度	M_8[*Addr*]
fstpl	*Addr*	Y	双精度	M_8[*Addr*]
fstt	*Addr*	N	扩展精度	M_{10}[*Addr*]
fstpt	*Addr*	Y	扩展精度	M_{10}[*Addr*]
fistl	*Addr*	N	整数	M_4[*Addr*]
fistpl	*Addr*	Y	整数	M_4[*Addr*]
fst	%st(*i*)	N	扩展精度	%st(*i*)
fstp	%st(*i*)	Y	扩展精度	%st(*i*)

图 3.32 浮点存储指令

所有的指令将结果从扩展精度格式转换成目标格式。带后缀“p”的指令将栈顶元素弹出栈。

练习题 3.26

为下面这段代码做如下假设，寄存器%eax 包含整数变量 x，而栈顶两个元素分别对应于变量 a 和 b。在方框中写出每条指令后栈的内容。



写出一个用 x、a 和 b 表示的 C 表达式，描述这段代码序列结束之后，栈顶元素的内容。

最后的浮点数据传送操作允许交换两个浮点寄存器的内容。指令 `fxch %st(i)` 交换浮点寄存器%st(0)和%st(i)的内容。不带参数的符号 fxch 等价于 fxch %st(1)，也就是，交换两个栈顶元素。

3.14.4 浮点算术指令

图 3.33 说明了一些最常见的浮点算术操作。第一组中的指令没有操作数。它们将某些常数数字的浮点表示压入栈中。对像π、e 和 $\log_2 10$ 这样的常数，也有类似的指令。第二组中的指令有一个操作数。这个操作数总是栈顶的元素，类似于假设的栈求值器中的 neg 操作，它们会用计算出的值取代这个元素。第三组中的指令有两个操作数。对每个这样的指令，都有关于如何指定操作数的许多不同的变种，待会儿会谈到。对不可交换操作，例如减法和除法，有前向（例如 fsub）和反向（例如 fsubr）两个版本，这样就可以按照两种顺序中的任一种来使用参数。

指　令	计　算
fldz	0
fld1	1
fabs	$\lvert Op \rvert$
fchs	$-Op$
fcos	$\cos Op$
fsin	$\sin Op$
fsqrt	$\sqrt{Op}$
fadd	$Op_1 + Op_2$
fsub	$Op_1 - Op_2$
fsubr	$Op_2 - Op_1$
fdiv	Op_1 / Op_2
fdivr	Op_2 / Op_1
fmul	$Op_1 \cdot Op_2$

图 3.33 浮点算术操作

每个双操作数操作都有多个变种。

在图 3.33 中，我们只给出了减法操作 fsub 的一种形式。实际上，这个操作有多个变种，如图 3.34 所示。这些指令都是计算两个操作数之差：$Op_1 - Op_2$，并将结果存放到某个浮点寄存器中。除

了为假设的栈求值器考虑的简单 subp 指令以外，IA32 还有一些指令是从存储器或某个除%st(1)以外的浮点寄存器中读出它们的第二个操作数的。另外，它们也都有弹出和不弹出这两个变种。第一组指令从存储器中读出第二个操作数，这个数可以是单精度、双精度或整数格式的。然后，它再把这个数转换成扩展精度格式，将栈顶元素减去这个数，并覆盖栈顶元素。这可以看成是一个浮点加载，后面跟一个基于栈的减法操作的组合。

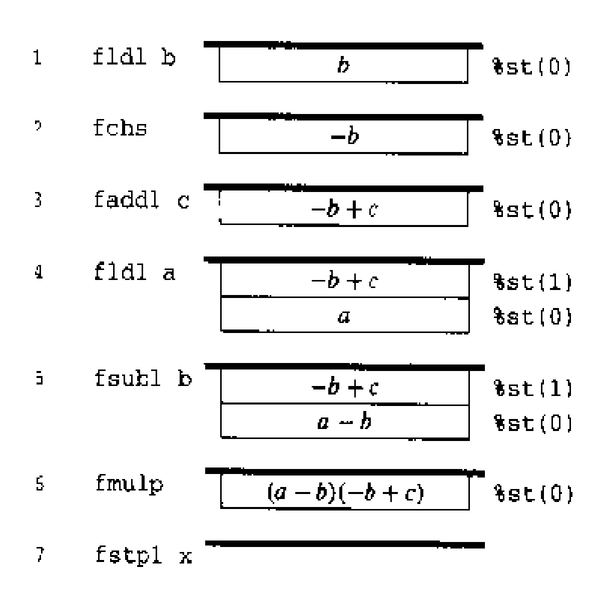
指令		操作数 1	操作数 2	格式	目的	弹出%st(0)(Y/N)
fsubs	*Addr*	%st(0)	$M_4[Addr]$	单精度	%st(0)	N
fsubl	*Addr*	%st(0)	$M_8[Addr]$	双精度	%st(0)	N
fsubt	*Addr*	%st(0)	$M_{10}[Addr]$	扩展精度	%st(0)	N
fisubl	*Addr*	%st(0)	$M_4[Addr]$	整数	%st(0)	N
fsub	%st(i),%st	%st(i)	%st(0)	扩展精度	%st(0)	N
fsub	%st(0),%st(i)	%st(0)	%st(i)	扩展精度	%st(i)	N
fsubp	%st(0),%st(i)	%st(0)	%st(i)	扩展精度	%st(i)	Y
fsubp		%st(0)	%st(i)	扩展精度	%st(1)	Y

图 3.34 浮点减法指令

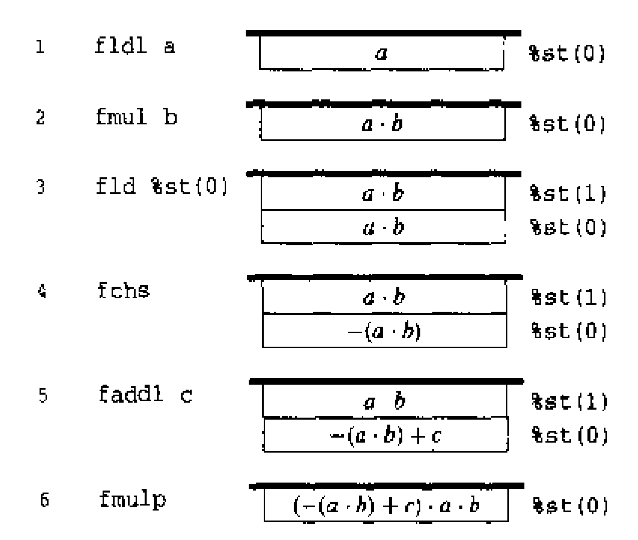
所有的指令都将结果以扩展精度格式存放到一个浮点寄存器中。带后缀“p”的指令会弹出栈顶元素。

第二组减法指令以栈顶元素作为一个参数，以另外一个栈元素作为另一个参数，但是它们的参数顺序、结果所使用的目的，以及是否会弹出栈顶元素都是不一样的。注意，汇编代码行 `fsubp` 是 `fsubp %st, %st(1)` 的简写。这一行对应于我们假设的栈求值器的 subp 指令。也就是，它计算栈顶两元素之差，将结果存放在%st(1)中，然后弹出%st(0)，这样计算出的值就在栈顶了。

图 3.33 中列出的所有双操作数操作，都有图 3.34 中列出的 fsub 的所有变种。例如，我们可以用 IA32 指令写出表达式 x = (a-b)*(-b+c)的代码。为了说明方便，我们仍然使用存储器位置的符号名字，并假设这些都是双精度值。

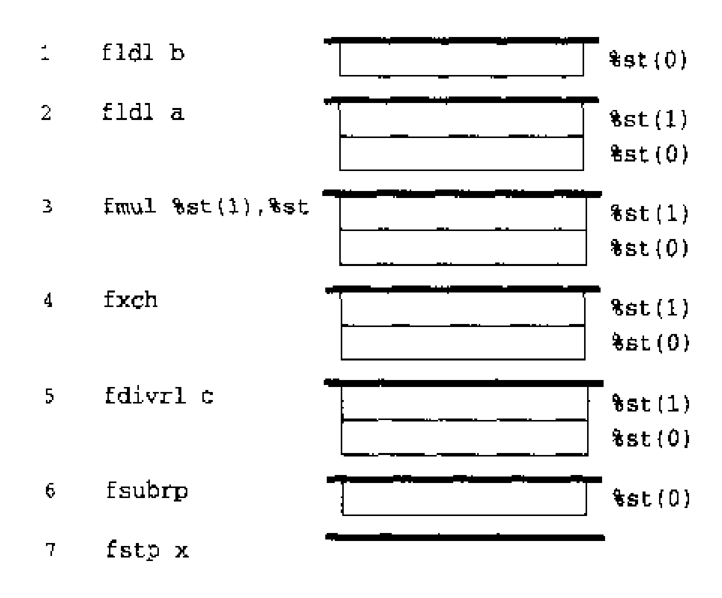


再来看一个例子，考虑表达式 x = (a*b)+(-(a*b)+c)。注意是如何用指令 `fld %st(0)` 在栈中创建 a*b 的两个副本的，这样避免了在临时存储器位置中保存这个值。



练习题 3.27

画出下述代码每一步之后栈的内容：



用一个 C 表达式来描述这个计算。

3.14.5 在过程中使用浮点

同整数参数一样，浮点参数是通过栈传递给调用过程的。每个 float 类型的参数需要 4 个字节的栈空间，而每个 double 类型的参数需要 8 个字节。对于返回值为 float 或 double 类型的函数，结果是以扩展精度格式在浮点寄存器栈顶部返回的。

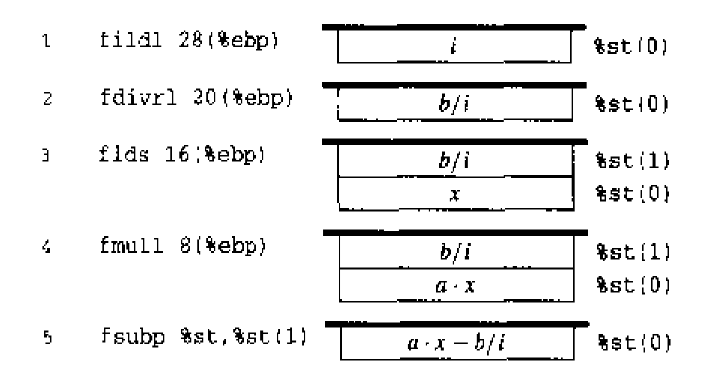
作为一个示例，看看下面这个函数：

```
1  double funct(double a, float x, double b, int i)
2  {
3      return a*x - b/i;
4  }
```

相对于%ebp，参数 a、x、b 和 i 的位置分别为 8、16、20 和 28：

偏移	8	16	20	28
内容	a	x	b	i

产生代码的主体，以及得到的栈值如下所示：



练习题 3.28

对于带参数 a、x、b 和 i（以及与 funct 不同的声明）的函数 funct2，编译器为函数体产生下面这样的代码：

```
1        movl 8(%ebp),%eax
2        fldl 12(%ebp)
3        flds 20(%ebp)
4        movl %eax,-4(%ebp)
5        fildl -4(%ebp)
6        fxch %st(2)
7        faddp %st,%st(1)
8        fdivrp %st,%st(1)
9        fld1
10       flds 24(%ebp)
11       faddp %st,%st(1)
```

返回值的类型为 double。写出 funct2 的 C 代码。注意要保证正确声明参数的类型。

3.14.6 测试和比较浮点值

类似于整数的情况，确定两个浮点数的相对值包括用比较指令来设置条件码，然后再测试这些条件码。不过，对于浮点，条件码是浮点状态字的一部分，浮点状态字是一个 16 位寄存器，包含关于浮点单元的各种标志。必须将这个状态字转换成整数字，然后测试某些特殊的位。

如图 3.35 所示，有很多不同的浮点比较指令。所有这些指令执行的都是操作数 Op_1 和 Op_2 之间的比较，这里 Op_1 是栈顶元素。表中每一行说明了两条不同的比较指令：一个是有序比较，用于像<和≤这样的比较；而另一个是无序比较，用于相等的比较。两种比较的区别只在于它们对待 *NaN* 值[3]是不同的，因为 *NaN* 值和其他值之间没有相对顺序。例如，如果变量 x 是一个 *NaN*，而变量 y 是某个其他值，那么表达式 $x < y$ 和 $x = y$ 都应该产生 0。

3 关于 *NaN* 的解释见 2.4.3 节的末尾。——译者

有序		无序		Op_2	类型	弹出元素的个数
fcoms	*Addr*	fucoms	*Addr*	$M_4[Addr]$	单精度	0
fcoml	*Addr*	fucoml	*Addr*	$M_8[Addr]$	双精度	0
fcom	%st(*i*)	fucom	%st(*i*)	%st(*i*)	扩展精度	0
fcom		fucom		%st(1)	扩展精度	0
fcomps	*Addr*	fucomps	*Addr*	$M_4[Addr]$	单精度	1
fcompl	*Addr*	fucompl	*Addr*	$M_8[Addr]$	双精度	1
fcomp	%st(*i*)	fucomp	%st(*i*)	%st(*i*)	扩展精度	1
fcomp		fucomp		%st(1)	扩展精度	1
fcompp		fucompp		%st(1)	扩展精度	2

图 3.35 浮点比较指令

有序和无序比较不同之处在于它们对待 *NaN* 值是不同的。

比较指令的各种形式的不同之处还在于操作数 Op_2 的位置是不同的，类似于浮点加载和浮点算术指令的各种形式。最后，各种形式的不同之处还在于，在比较完成后从栈中弹出的元素的个数。表中所示的第一组指令根本不会改变栈。即使是对于一个参数在存储器中的情况，最终这个值也不会放在栈中。第二组中的操作会将元素 Op_1 弹出栈。而最后一个操作则会将 Op_1 和 Op_2 都弹出栈。

指令 fnstsw 将浮点状态字传送到一个整数寄存器。这条指令的操作数是图 3.2 中所示的 16 位寄存器标识符中的一个，例如%ax。状态字中，对比较结果编码的位是状态字的高位字节的 0、2 和 6 位。例如，如果我们用指令 fnstw %ax 传送状态字，那么相应的位就在%ah 中。选择这些位的典型代码序列是这样的：

```
1    fnstsw %ax       Store floating point status word in %ax
2    andb $69,%ah     Mask all but bits 0, 2, and 6
```

注意，69_{10} 的位表示为[00100101]，也就是，三个相应位上的值均为 1。图 3.36 给出了由这段代码序列得到的字节%ah 可能的值。注意，对于比较操作数 Op_1 和 Op_2 只有四种可能的结果：第一个数大于、小于、等于第二个数，或是两者不能比较，只有当一个值为 *NaN* 时，才会出现最后一种结果。

Op_1: Op_2	二进制	十进制
>	[00000000]	0
<	[00000001]	1
=	[00100000]	64
无序	[00100101]	69

图 3.36 对浮点比较结果的编码

结果编码在浮点状态字的高位字节，屏蔽了除 0、2 和 6 以外的其他位。

看看下面这个过程示例：

```
1  int less(double x, double y)
2  {
```

```
3       return x < y;
4   }
```

这个函数体的编译后代码是这样的：

```
1    fldl 16(%ebp)          Push y
2    fcompl 8(%ebp)         Compare y:x
3    fnstsw %ax             Store floating point status word in %ax
4    andb $69,%ah           Mask all but bits 0, 2, and 6
5    sete %al               Test for comparison outcome of 0 (>)
6    movzbl %al,%eax        Copy low order byte to result, and set rest to 0
```

练习题 3.29

请说明，如何通过在前面的代码序列中插入一行汇编代码，就能实现下面的函数：

```
1   int greater(double x, double y)
2   {
3       return x > y;
4   }
```

现在，我们就讲完了用 IA32 进行汇编级浮点编程。即使是有经验的程序员也会觉得这些代码很神秘，难以阅读。基于栈的操作，将状态结果从 FPU 读到主处理器的笨拙，以及浮点计算的许多细微之处，都使得机器代码冗长而晦涩。值得注意的是，如果数字程序被编码为指定格式，则 Intel 和它的竞争者们生产的现代处理器就能够使这些数字程序达到相当高的性能。

3.15 *在 C 程序中嵌入汇编代码

在早期的计算中，大多数程序都是用汇编代码写的，即使是很大型的操作系统也是在没有高级语言帮助的情况下编写的。就程序的复杂性来说，这就变得难以管理了。因为汇编代码不提供任何形式的类型检查，所以很容易犯基本的错误，例如将指针作为整数来用，而不是间接引用指针。更糟的是，用汇编写代码会将整个程序限制在某一类机器上了。重写一个汇编语言程序，使它能在不同的机器上运行，与从头写整个程序是一样困难的。

旁注：用汇编代码编写大型程序

Frederick Brooks, Jr.，一位计算机系统的先驱，编写了关于 OS/360 开发的说明。OS/360 是 IBM 机器的一个早期操作系统[5]，直到今天它还提供了很多重要的经验。通过写这些东西，他成为了用高级语言进行系统编程的衷心拥护者。不过，令人惊奇的是，有一组活跃的程序员，他们很高兴为 IA32 写汇编代码。他们通过 Internet 新闻组 comp.lang.asm.x86 来彼此联系。他们中的大多数为 DOS 操作系统编写计算机游戏。

早期的高级编程语言的编译器不能产生非常有效的代码，也不能提供系统程序员常常需要的对低级目标（代码）表示的访问。要求高性能或需要访问目标（代码）表示的程序通常还是用汇编代码来写的。不过现在，优化编译器基本上使得性能优化不再是用汇编代码写程序的一个原因了。一个高质量的编译器产生的代码通常和手工编写的一样好，甚至于更好。而 C 语言基本上使得机器访

问不再使用汇编代码了。C 语言能够通过联合和指针运算访问低级数据表示，以及能对位级数据表示进行操作，这就为大多数程序员提供了足够多访问机器的能力。例如，像 Linux 这样的现代操作系统，几乎每个部分都是用 C 写的。

尽管如此，有时候用汇编写代码仍然是惟一的选择，特别是实现操作系统时就更是这样。比如说，操作系统必须访问一些特殊的寄存器，它们存放着进程状态信息，执行输入和输出操作要使用特殊的指令或是访问特殊的存储器位置。即使是对应用程序员来说，也有一些机器特性，例如条件码的值，是不能直接用 C 访问的。

现在的问题是要将主要由 C 组成的代码与少量汇编代码集成到一起。一种方法是用汇编代码写一些关键函数，使用的参数传递和寄存器使用规则与 C 编译器遵守的一样。这些汇编函数保存在独立的文件中，由链接器将编译好的 C 代码和汇编好的汇编代码结合起来。例如，如果文件 p1.c 包含 C 代码，而文件 p2.s 包含的是汇编代码，那么编译命令

```
unix> gcc -o p p1.c p2.s
```

会编译文件 p1.c 和汇编文件 p2.s，并将得到的目标代码链接形成可执行程序 p。

3.15.1 基本的内嵌汇编（inline assembly）

GCC 还可以将汇编与 C 代码混合起来。内嵌汇编允许用户直接往编译器产生的代码序列中插入汇编代码。可以提供一些特性，以指定指令操作数和向编译器说明汇编指令要覆盖哪些寄存器。当然，得到的代码是与机器高度相关的，因为不同类型机器的机器指令是不兼容的、asm 命令（directive）也是与 GCC 相关的，它与很多其他编译器是不兼容的。尽管如此，这还是一种有效的方式，将与机器相关的代码数量降低到绝对小。

内嵌汇编是作为 GCC 信息档案的一部分来说明的，在任何安装了 GCC 的机器上执行命令 info gcc，会得到一个分层的文档阅读器。沿着名为“C Extensions”的链接，然后是名为“Extended Asm”的链接，就能找到内嵌汇编的文档。不幸的是，这个文档有点不完全，也不太准确。

内嵌汇编的基本格式是像过程调用一样写代码：

```
asm( code-string );
```

术语 code-string 表示一个以带括号的字符串形式给出的汇编代码序列。编译器会将这个字符串一字不差地插入到产生的汇编代码中，因此，编译器提供的汇编和用户提供的汇编就合并到一起了。编译器不会检查字符串是否出错，因此，要等到汇编器才会报告错误。

我们以一个需要访问条件码的例子来说明 asm 的使用。考虑原型如下的函数：

```
int ok_smul(int x, int y, int *dest);

int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

每个函数都用来计算参数 x 和 y 的乘积，并将结果存放到参数 dest 指定的存储器位置中。至于返回值，当乘法溢出时会返回 0，否则返回 1。有符号乘和无符号乘是两个函数，因为它们的溢出情况是不同的。

分析 IA32 乘法指令 mul 和 imul 的文档，我们看到在溢出时，两个指令都会设置进位标志 CF。查看图 3.10，我们看到指令 setae 可以用来在 CF 标志设为 1 时，将一个寄存器的低位字节设置为 0，否则就设置为 1。因此，我们希望将这条指令插入到编译器产生的序列中。

在试图使用尽可能少的汇编代码和详细分析后，我们试着用下面的代码来实现ok_smul：

code/asm/okmul.c

```
1   /* First attempt. Does not work */
2   int ok_smul1(int x, int y, int *dest)
3   {
4       int result = 0;
5
6       *dest = x*y;
7       asm("setae %al");
8       return result;
9   }
```

code/asm/okmul.c

这里的策略利用的是寄存器%eax 是用来存放返回值的。假设编译器用这个寄存器来存放变量result，第一行就会将这个寄存器设置为0。内嵌汇编会插入正确设置这个寄存器低位字节的代码，而这个寄存器会用来作为返回值。

不幸的是，GCC有它自己的关于代码产生的想法。产生的代码并不会在函数一开始时就将寄存器%eax 设置为0，而是到最后才这么做，所以函数总是返回0。最根本的问题是，编译器无法知道程序员的意图是什么，也无法知道汇编语句应该如何与其他产生的代码交互。

通过一系列尝试（待会儿我们会详细介绍更加系统的方法），我们能生成可行的代码，但是这也不太理想：

code/asm/okmul.c

```
1   /* Second attempt. Works in limited contexts */
2   int dummy = 0;
3
4   int ok_smul2(int x, int y, int *dest)
5   {
6       int result;
7
8       *dest = x*y;
9       result = dummy;
10      asm("setae %al");
11      return result;
12  }
```

code/asm/okmul.c

这段代码使用的是和前面一样的策略，但是它用全局变量dummy的值来将result初始化为0。对于产生包含全局变量的代码，编译器通常会比较保守，所以不太可能会重新排列计算的顺序。

前面的代码依赖于编译器能够处理得当。实际上，只有当编译器的优化选项（命令行选项-O）是打开的时候，这段代码才能正常工作。当不带优化编译时，它会将result存放在栈中，在返回之前取出，覆盖setae指令设置的值。编译器无法知道插入的汇编语言与其他代码之间的关系，因为我们没有提供给编译器这样的信息。

3.15.2 asm 的扩展格式

GCC 提供了 asm 的一个扩展版本，它允许程序员指定哪些程序值要作为汇编代码序列的操作数，以及哪些寄存器要被汇编代码覆盖。有了这些信息，编译器产生的代码就能正确建立所需要的源值，执行汇编指令，并使用计算出的值。这些信息中还包括编译器所需的关于寄存器使用的信息，这样一来，重要的程序值就不会被汇编代码指令覆盖了。

扩展的汇编序列的通用语法是这样的：

asm(*code-string* [: *output-list* [: *input-list* [: *overwrite-list*]]]);

这里，方括号表示可选参数。这个声明包含一个描述汇编代码序列的字符串，后面是可选的列表，包括输出（也就是汇编代码产生的结果）、输入（也就是汇编代码的源值），以及汇编代码会覆盖的寄存器。这些列表以冒号（:）分隔。正如方括号表明的那样，我们只包含到最后一个非空的列表。

代码串的语法让人想起 printf 语句中格式化字符串的语法。它是由一个用分号（“;”）分隔的汇编代码指令序列组成的。输入和输出操作数由引用%0，%1，…，%9 表示。操作数是根据它们第一次在输出列表和输入列表中出现的顺序编号的。像“%eax”这样的寄存器名字必须要多加一个“%”符号，也就是写成“%%eax”。

下面是 ok_smul 的一个更好的实现，它使用扩展的汇编语句来告诉编译器汇编语句是为变量 result 产生的值：

code/asm/okmul.c

```
1   /* Uses extended asm to get reliable code */
2   int ok_smul3(int x, int y, int *dest)
3   {
4       int result;
5
6       *dest = x*y;
7
8       /* Insert the following assembly code:
9          setae %bl                 # Set low-order byte
10         movzbl %bl, result        # Zero extend to be result
11      */
12      asm("setae %%bl; movzbl %%bl,%0"
13          : "=r" (result)          /* Output */
14          :                        /* No inputs */
15          : "%ebx"                 /* Overwrites */
16          );
17
18      return result;
19  }
```

code/asm/okmul.c

第一条汇编指令将测试结果保存在单字节寄存器%bl 中。然后，第二条指令对这个值进行零扩展，并拷贝到编译器选择的用来保存 result 的随便哪个寄存器中，result 是用操作数%0 表示的。输

出列表是由以空格分隔的值对组成的。(在本例中，只有一个值对。)值对的第一个元素是一个字符串，表明操作数的类型，这里“r”表示一个整数寄存器，而“=”表示汇编代码对这个操作数进行了赋值。值对的第二个元素是用括号括起来的操作数。它可以是任何可赋值的值(在C中称为左值，lvalue)。编译器会产生必要的代码序列来执行这个赋值。输入列表有相同的通用格式，这里操作数可以是任意C表达式。编译器会产生必要的代码来对这个表达式求值。覆盖列表只是简单地给出会被重写的寄存器的名字(作为带括号的字符串)。

无论编译选项如何，前面这段代码都能正常工作。正如这个示例表明的那样，要编写允许操作数按照要求的格式书写的汇编代码，可能还需要一点点创造性的思维。例如，没有直接的方法来指定一个程序值作为 setae 指令的目的操作数，因为这个操作数必须是单字节的。[4] 因此，我们编写了一个基于一个特殊寄存器的代码序列，然后用一个额外的数据传送指令来将得到的值拷贝到程序状态的某个部分。

练习题 3.30

GCC 提供了扩展精度运算的工具。它可以用来实现 ok_smul 函数，优点是函数可以跨机器移植。声明为类型“long long”的变量的大小为普通 long 变量的两倍。因此，语句

```
long long prod = (long long) x * y;
```

会计算 x 和 y 的全 64 位乘积。用这个工具，写出一个不使用任何 asm 语句的 ok_smul 版本。

有人可能会想，这段代码序列可以用在 ok_umul 中，但是，对有符号和无符号乘法，GCC 用的都是 imull (有符号乘法) 指令。虽然它能为两个乘法都产生正确的值，但是它会根据有符号乘法的规则来设置进位标志。因此，我们需要使用汇编代码序列，显式地用图 3.9 中说明的 mull 指令来执行无符号乘法，这段代码如下所示：

code/asm/okmul.c

```
1   /* Uses extended asm */
2   int ok_umul(unsigned x, unsigned y, unsigned *dest)
3   {
4       int result;
5
6       /* Insert the following assembly code:
7          movl x,%eax           # Get x
8          mull y                # Unsigned multiply by y
9          movl %eax,*dest       # Store low-order 4 bytes at dest
10         setae %dl             # Set low-order byte
11         movzbl %dl, result    # Zero extend to be result
12       */
13       asm("movl %2,%%eax; mull %3; movl %%eax,%0;
14            setae %%dl; movzbl %%dl,%1"
15          : "=r" (*dest), "=r" (result)     /* Outputs */
```

4 实际上，你可以用 GCC 声明一个类型为 char 的变量来声明一个单字节操作数，参见 http://www.csapp.cs.cmu.edu/public/byteasm.html。——译者

```
16              : "r" (x), "r" (y)                    /* Inputs */
17              : "%eax", "%edx"                      /* Overwrites */
18              );
19
20          return result;
21      }
```

code/asm/okmul.c

回忆一下，mull 指令要求它的一个参数在寄存器%eax 中，而第二个参数是作为操作数给出的。为了说明这一点，在 asm 语句中，我们用 movl 将程序值 x 传送到%eax，并指明程序值 y 是 mull 指令的参数。然后，指令会将 8 个字节的乘积存放在两个寄存器中，%eax 保存低位 4 个字节，而%edx 保存高位字节。然后我们用寄存器%edx 来构造返回值。正如这个示例说明的那样，逗号（,）用来在输入和输出列表中分隔操作数对，以及在覆盖列表中分隔寄存器名字。注意，我们能够将*dest 指定为第二个 movl 指令的输出，因为它是可赋值的。于是，编译器会产生正确的机器代码，将%eax 中的值存储在这个存储器位置上。

想了解编译器是如何产生关于 asm 语句的代码的，下面是为 ok_umul 产生的代码：

```
    Set up asm inputs
1       movl 8(%ebp),%ecx           Load x into %ecx
2       movl 12(%ebp),%ebx          Load y into %ebx
3       movl 16(%ebp),%esi          Load dest into %esi
     The following instructions were generated by asm.
       Input registers: %ecx for x, %ebx for y
       Output registers: %ecx for product, %ebx for result
4       movl %ecx,%eax; mull %ebx; movl %eax,%ecx;
5       setae %dl; movzbl %dl,%ebx
     Process asm outputs
6       movl %ecx,(%esi)            Store product at dest
7       movl %ebx,%eax              Set result as return value
```

这段代码的第 1～3 行取出过程参数，并将它们存放到寄存器中。注意，它没有使用寄存器%eax 或%edx，因为我们已经声明了这两个寄存器会被重写。第 4 行和第 5 行是我们的内嵌汇编代码，不过参数换成了寄存器的名字。特别地，它会用寄存器%ecx 代替参数%2（x），用%ebx 代替参数%3（y）。乘积会暂时存放在%ecx 中，而它会用寄存器%ebx 代替参数%1（result）。然后，第 6 行将乘积存储到 dest，完成了对参数%0（*dest）的处理。第 7 行将 result 拷贝到寄存器%eax，作为返回值。因此，编译器不仅产生了我们 asm 语句指示的代码，还产生了提供语句输入（第 1～3 行）和使用输出（第 6～7 行）的代码。

虽然 asm 语句的语法有点难懂，而且它的使用也使代码的可移植性变差了，但是对于编写用少量汇编代码来访问机器级特性的程序，这条语句还是非常有用的。我们发现，要想代码能正常工作，是需要进行一些尝试和犯点错误的。最好的办法就是用-S 选项编译选项，然后检查产生出的汇编代码，看它是否达到了期望的效果。代码还应该用不同的选项设置来测试，例如带和不带-O 选项。

3.16 小结

在本章中，我们窥视了高级语言提供的抽象层下面的东西，以了解机器级编程。通过让编译器产生机器级程序的汇编代码表示，我们了解了编译器和它的优化能力，以及机器代码、它的数据类型和它的指令集。在第5章中，我们会看到，当编写能有效映射到机器上的程序时，了解编译器的特性会有所帮助。我们还看了一些高级语言抽象隐藏有关程序操作重要细节的例子。例如，浮点代码的行为可能依赖于值是保存在寄存器中，还是在存储器中。在第13章中，我们会看到许多这样的例子，我们需要知道一个程序变量是在运行时栈中，是在某个动态分配的数据结构中，还是在某个全局存储位置中。理解程序是如何映射到机器上的，会让理解这些存储之间的区别容易一些。

汇编语言与C代码差别很大。在汇编语言程序中，各种数据类型之间的差别很小。程序是以指令序列来表示的，每条指令都完成一个单独的操作。部分程序状态，如寄存器和运行时栈，对程序员来说是直接可见的。仅提供了低级操作来支持数据处理和程序控制。编译器必须用多条指令来产生和操作各种数据结构，来实现像条件、循环和过程这样的控制结构。我们讲述了C和如何编译C的许多不同方面。我们看到C中缺乏边界检查，使得许多程序容易出现缓冲区溢出，而这已经使许多系统容易受到入侵者的恶意攻击。

我们只分析了C到IA32的映射，但是我们讲的大多数内容对其他语言和机器组合来说也是类似的。例如，编译C++与编译C就非常相似。实际上，C++的早期实现就只是简单地执行了从C++到C的源到源的转换，并对结果运行C编译器，产生目标代码。C++的对象用结构来表示，类似于C的struct。C++的方法是用指向实现方法的代码的指针来表示的。相比而言，Java的实现方式完全不同。Java的目标代码是一种特殊的二进制表示，称为Java字节代码。这种代码可以看成是虚拟机的机器级程序。正如它的名字暗示的那样，这种机器并不是直接用硬件实现的。相反，软件解释器处理字节代码，模拟虚拟机的行为。这种方法的优点是相同的Java字节代码可以在许多不同的机器上执行，而我们在本章谈到的机器代码只能在IA32上运行。

参考文献说明

关于IA32最好的参考书目来自于Intel。他们关于软件开发的系列中有两本特别有用。基本体系结构手册[18]给出了从汇编语言程序员角度来看的体系结构概貌，而指令集参考手册[19]给出了各种指令的详细描述。这些参考书目包含的信息远远超出了理解Linux代码所需要的内容。特别地，Linux使用平面模式寻址，所有分段寻址方法的复杂性都可以不予考虑了。

Linux汇编器使用的GAS格式与Intel文档中以及其他编译器(特别是Microsoft生产的编译器)使用的标准格式差别很大。一个主要区别就是源和目的操作数是以相反的顺序给出的。

在Linux机器上，运行命令info as会显示有关汇编器的信息。其中一个小部分说明了与机器相关的信息，包括GAS与更标准的Intel表示法的比较，注意，GCC称这些机器为“i386”——它产生的代码甚至于可以在1985年的机器上运行。

Muchnick的关于编译器设计的著作[55]被认为是有关代码优化技术最全面的参考文献。它覆盖了我们在此讨论的许多技术，例如寄存器使用规则和基于do-while格式为循环产生代码的优点。

关于通过Internet用缓冲区溢出来攻击系统，已经有很多论述了。Spafford[73]出版了关于1988年Internet蠕虫的详细分析，帮助制止这种蠕虫传播的MIT的一些人也出版了一些论著[26]。从那

以后，产生了许多关于制造和预防缓冲区溢出攻击的论文和项目，例如[20]。

家庭作业

3.31 ◆

给你如下信息。一个函数的原型为

```
int decode2(int x, int y, int z);
```

将这个函数编译成汇编代码。代码体为：

```
1      movl 16(%ebp),%eax
2      movl 12(%ebp),%edx
3      subl %eax,%edx
4      movl %edx,%eax
5      imull 8(%ebp),%edx
6      sall $31,%eax
7      sarl $31,%eax
8      xorl %edx,%eax
```

参数 x、y 和 z 存放在存储器中相对于寄存器%ebp 中地址偏移量为 8、12 和 16 的地方。代码将返回值存放在寄存器%eax 中。

写出等价于我们汇编代码 decode2 的 C 代码。可以通过用-S 选项编译你的代码来测试你的答案。你的编译器产生的代码不一定完全一样，但是功能应该等价。

3.32 ◆◆

下面的 C 代码基本上与图 3.12 中的代码相同：

```
1  int absdiff2(int x, int y)
2  {
3      int result;
4
5      if (x < y)
6         result = y-x;
7      else
8         result = x-y;
9      return result;
10 }
```

不过编译时，它得到形式不同的汇编代码：

```
1      movl 8(%ebp),%edx
2      movl 12(%ebp),%ecx
3      movl %edx,%eax
4      subl %ecx,%eax
5      cmpl %ecx,%edx
6      jge .L3
7      movl %ecx,%eax
8      subl %edx,%eax
9    .L3:
```

A. 当x<y时，执行哪个减法？当x≥y时呢？

B. 这段代码与前面讲过的if-else的标准实现有什么不同？

C. 用C语法（包括goto），给出这个翻译的通用形式。

D. 为了保证这个翻译具有C代码指定的行为，要对它的使用加上什么样的限制？

3.33 ◆◆

下面的代码给出了一个开关语句中根据枚举类型值进行分支选择的例子。回忆一下，C中枚举类型只是一种引入一组与整数值相对应的名字的方法。默认情况下，值是从0向上依次赋给名字的。在我们的代码中，省略了与各种情况标号（case labels）相对应的动作。

```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
{
  int result = 0;
  switch(action) {
  case MODE_A:

  case MODE_B:

  case MODE_C:

  case MODE_D:

  case MODE_E:

  default:

  }
  return result;
}
```

产生的实现各个动作的汇编代码部分如图3.37所示。注释表明了存储在寄存器中的值，以及各个跳转目的的情况标号。

A. 程序变量result对应于哪个寄存器？

B. 填写出C代码中缺失的部分。注意会落入其他情况（cases）的情况（case）。

```
     The jump targets
     Arguments p1 and p2 are in registers %ebx and %ecx.
1    .L15:                          MODE_A
2      movl (%ecx),%edx
3      movl (%ebx),%eax
4      movl %eax,(%ecx)
5      jmp .L14
6      .p2align 4,,7            Inserted to optimize cache performance
7    .L16:                          MODE_B
8      movl (%ecx),%eax
```

```
9      addl (%ebx),%eax
10     movl %eax,(%ebx)
11     movl %eax,%edx
12     jmp .L14
13     .p2align 4,,7          Inserted to optimize cache performance
14   .L17:                        MODE_C
15     movl $15,(%ebx)
16     movl (%ecx),%edx
17     jmp .L14
18     .p2align 4,,7          Inserted to optimize cache performance
19   .L18:                        MODE_D
20     movl (%ecx),%eax
21     movl %eax,(%ebx)
22   .L19:                        MODE_E
23     movl $17,%edx
24     jmp .L14
25     .p2align 4,,7          Inserted to optimize cache performance
26   .L20:
27     movl $-1,%edx
28     .L14:                  default
29     movl %edx,%eax         Set return value
```

图 3.37 作业 3.33 的汇编代码

这段代码实现了 switch 语句的各个分支。

3.34 ◆◆

对于从目标代码进行逆向工程来说，开关语句是特别困难的。在下面这个过程中，去掉了开关语句的主体：

```
1   int switch_prob(int x)
2   {
3     int result = x;
4
5     switch(x) {
6
7         /* Fill in code here */
8     }
9
10    return result;
11  }
```

图 3.38 给出的是这个过程的反汇编目标代码。我们只对第 4～16 行所示的代码部分感兴趣。在第 4 行我们可以看到参数 x（在相对于%ebp 偏移量为 8 的位置）被加载到寄存器%eax 中，对应于程序变量 result。第 11 行的指令 `lea 0x0(%esi), %esi` 是一条空指令，插入这条指令是为了使第 12 行的指令的起始地址为 16 的倍数。

```
1   080483c0 <switch_prob>:
2    80483c0: 55                      push   %ebp
3    80483c1: 89 e5                   mov    %esp,%ebp
4    80483c3: 8b 45 08                mov    0x8(%ebp),%eax
5    80483c6: 8d 50 ce                lea    0xffffffce(%eax),%edx
6    80483c9: 83 fa 05                cmp    $0x5,%edx
7    80483cc: 77 1d                   ja     80483eb <switch_prob+0x2b>
8    80483ce: ff 24 95 68 84 04 08    jmp    *0x8048468(,%edx,4)
9    80483d5: c1 e0 02                shl    $0x2,%eax
10   80483d8: eb 14                   jmp    80483ee <switch_prob+0x2e>
11   80483da: 8c b6 00 00 00 00       lea    0x0(%esi),%esi
12   80483e0: c1 f8 02                sar    $0x2,%eax
13   80483e3: eb 09                   jmp    80483ee <switch_prob+0x2e>
14   80483e5: 8d 04 40                lea    (%eax,%eax,2),%eax
15   80483e8: 0f af c0                imul   %eax,%eax
16   80483eb: 83 c0 0a                add    $0xa,%eax
17   80483ee: 89 ec                   mov    %ebp,%esp
18   80483f0: 5d                      pop    %ebp
19   80483f1: c3                      ret
20   80483f2: 89 f6                   mov    %esi,%esi
```

图 3.38 作业 3.34 的反汇编代码

跳转表在另一块存储器区域中。用调试器 GDB，我们可以用命令 x/6w 0x8048468 来检查存储器中从地址 0x8048468 开始的六个 4 字节的字。GDB 打印出下面的内容：

```
(gdb) x/6w 0x8048468
0x8048468:  0x080483d5  0x080483eb  0x080483d5  0x080483e0
0x8048478:  0x080483e5  0x080483e8
(gdb)
```

用 C 代码填写出开关语句的主体，使它的行为与目标代码一致。

3.35 ◆◆

C 编译器为 var_prod_ele 产生的代码（图 3.25(b)）不是最优的。根据过程 fix_prod_ele_opt（图 3.24）和 var_prod_ele_opt（图 3.25），写出这个函数的代码，使之对 n 的所有值都正确，但是编译成的代码要将它的所有临时数据都放在寄存器中。

回忆一下，处理器只有六个寄存器可用来保存临时数据，因为寄存器%ebp 和%esp 不能用于此目的。其中一个寄存器还必须用来保存乘法指令的结果。因此，你必须把循环中的局部变量的数量从六（result、Aptr、B、nTjPk、n 和 cnt）减少到五。

3.36 ◆◆

如果你负责维护一个大型的 C 程序，遇到下面这样的代码：

```
1  typedef struct {
2      int left;
3      a_struct a[CNT];
```

```
4       int right;
5   } b_struct;
6
7   void test(int i, b_struct *bp)
8   {
9       int n = bp->left + bp->right;
10      a_struct *ap = &bp->a[i];
11      ap->x[ap->idx] = n;
12  }
```

不幸的是，你对定义编译时常数 CNT 和结构 a_struct 的“.h”文件没有访问权限。幸好，你能够访问代码的“.o”版本，可以用 objdump 程序来反汇编这些文件，得到如图 3.39 所示的反汇编代码。

```
1 00000000  <test>:
2    0:   55                     push    %ebp
3    1:   89 e5                  mov     %esp,%ebp
4    3:   53                     push    %ebx
5    4:   8b 45 08               mov     0x8(%ebp),%eax
6    7:   8b 4d 0c               mov     0xc(%ebp),%ecx
7    a:   8d 04 80               lea     (%eax,%eax,4),%eax
8    d:   8d 44 81 04            lea     0x4(%ecx,%eax,4),%eax
9   11:   8b 10                  mov     (%eax),%edx
10  13:   c1 e2 02               shl     $0x2,%edx
11  16:   8b 99 b8 00 00 00      mov     0xb8(%ecx),%ebx
12  1c:   03 19                  add     (%ecx),%ebx
13  1e:   89 5c 02 04            mov     %ebx,0x4(%edx,%eax,1)
14  22:   5b                     pop     %ebx
15  23:   89 ec                  mov     %ebp,%esp
16  25:   5d                     pop     %ebp
17  26:   c3                     ret
```

图 3.39　作业 3.36 的反汇编代码

运用你的逆向工程技能，推断出下列内容：

A. CNT 的值。

B. 结构 a_struct 的完整声明。假设这个结构中只有域 idx 和 x。

3.37 ◆

编写一个函数 good_echo，它从标准输入读入一行，再写回到标准输出。你的实现必须对任意长度的输入行都能正常工作。可以使用库函数 fgets，但是必须保证，你的函数即使是在输入行需要比你为缓冲区分配的空间更大的空间时，仍能正确工作。你的代码还应该检查出错条件，当遇到错误时返回。关于标准 I/O 函数的定义可以参考文档[32，40]。

3.38 ◆◆◆

在这个问题中，你要着手对你自己的程序进行缓冲区溢出攻击。前面我们说过，我们不能原谅

用这种或其他形式的攻击来获得对系统的未被授权的访问，但是通过这个练习，你会学到许多关于机器级编程的知识。

从CS:APP的网站上下载文件bufbomb.c，编译它创建一个可执行文件。在bufbomb.c中，你会发现下面的函数：

```
1   int getbuf()
2   {
3      char buf[12];
4      getxs(buf);
5      return 1;
6   }
7
8   void test()
9   {
10    int val;
11    printf("Type Hex string:");
12    val = getbuf();
13    printf("getbuf returned 0x%x\n", val);
14  }
```

函数getxs（也在bufbomb.c中）类似于库函数gets，除了它是以十六进制数字对的编码方式读入字符的以外。比如说，要给它一个字符串“0123”，用户应该输入字符串“30 31 32 33”。这个函数会忽略空格字符。回忆一下，十进制数字x的ASCII表示为0x3x。

这个程序的典型执行是这样的：

```
unix> ./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

看看getbuf函数的代码，看上去似乎很明显，无论何时被调用，它都会返回值1。看上去就好像调用getxs没有产生效果一样。你的任务是，只简单地对提示符输入一个适当的十六进制字符串，就使getbuf对test返回-559038737（0xdeadbeef）。

下面这些建议可能会帮助你解决这个问题：

- 用OBJDUMP创建bufbomb的一个反汇编版本。仔细研究，确定getbuf的栈帧是如何组织的，以及溢出的缓冲区会如何改变保存的程序状态。
- 在GDB下运行你的程序。在getbuf中设置一个断点，并运行到该断点。确定像%ebp的值这样的参数，以及已保存的当缓冲区溢出时会被覆盖的所有状态的值。
- 手工确定指令序列的字节编码是很枯燥的，而且容易出错。可以用工具来完成这些工作，写一个汇编代码文件，包含想要放入栈中的指令和数据。用GCC汇编这个文件，再用OBJDUMP反汇编它，就可以获得要在提示符处输入的字节序列了。当OBJDUMP试图反汇编你文件中的数据时，它会产生一些看上去非常奇怪的指令，但是十六进制字节序列应该是正确的。

要记住，你的攻击是非常依赖于机器和编译器的。当运行在不同的机器上或使用不同版本GCC时，可能需要改变你的字符串。

3.39 ◆◆

用 asm 语句来实现一个具有如下原型的函数：

```
void full_umul(unsigned x, unsigned y, unsigned dest[]);
```

这个函数要计算它的参数的全 64 位乘积，并将结果存储到目的数组中，dest[0]存放低位 4 个字节，而 dest[1]存放高位 4 个字节。

3.40 ◆◆

fscale 指令计算浮点值 x 和 y 的函数 $x \cdot 2^{RTZ(y)}$，这里 RTZ 表示向 0 舍入（round-toward-zero）函数，将正数向下舍入，而将负数向上舍入。fscale 的参数来自于浮点寄存器栈，x 在%st(0)中，而 y 在%st(1)中。它将计算出来的值写入%st(0)，不弹出第二个参数。（这个指令的实际实现就是将 RTZ(y) 加到 x 的指数。）

用 asm 实现一个函数，它的原型为

```
double scale(double x, int n, double *dest);
```

它用 fscale 指令来计算 $x \cdot 2^n$，并将结果保存到由指针 dest 指定的位置。扩展的 asm 对 IA32 浮点的支持不是很好。不过，在这种情况中，你可以从程序栈中访问参数。

练习题答案

练习题 3.1 答案

这个练习使你熟悉各种操作数格式。

操作数	值	注释
%eax	0x100	寄存器
0x104	0XAB	绝对地址
$0x108	0x108	立即数
(%eax)	0XFF	地址 0x100
4(%eax)	0XAB	地址 0x104
9(%eax,%edx)	0x11	地址 0x10C
260($ecx,%edx)	0x13	地址 0x108
0XFC(,%ecx,4)	0xFF	地址 0x100
(%eax,%edx,4)	0x11	地址 0x10C

练习题 3.2 答案

逆向工程是一种理解系统的好方法。因此，我们想要逆转 C 编译器的效果，来确定什么样的 C 代码会得到这样的汇编代码。最好的方法是进行“模拟”，开始时，值 x、y 和 z 本别在指针 xp、yp 和 zp 指定的位置。于是，我们可以得到下面这样的效果：

```
1      movl 8(%ebp),%edi     xp
2      movl 12(%ebp),%ebx    yp
3      movl 16(%ebp),%esi    zp
```

```
4    movl (%edi),%eax      x
5    movl (%ebx),%edx      y
6    movl (%esi),%ecx      z
7    movl %eax,(%ebx)      *yp = x
8    movl %edx,(%esi)      *zp = y
9    movl %ecx,(%edi)      *xp = z
```

由此我们可以产生下面这样的 C 代码：

code/asm/decode1-ans.c

```
1  void decode1(int *xp, int *yp, int *zp)
2  {
3      int tx = *xp;
4      int ty = *yp;
5      int tz = *zp;
6
7      *yp = tx;
8      *zp = ty;
9      *xp = tz;
10 }
```

code/asm/decode1-ans.c

练习题 3.3 答案

这个练习说明了 leal 指令的多样性，同时也让你练习解读各种操作数形式。注意，虽然在图 3.3 中有的操作数格式被划分为“存储器”类型，但是并没有访存发生：

表达式	结果
`leal 6(%eax),%edx`	$6+x$
`leal (%eax %ecx ),%edx`	$x+y$
`leal (%eax %ecx,4 ),%edx`	$x+4y$
`leal 7(%eax %eax,8 ),%edx`	$7+9x$
`leal 0xA(, %ecx,4 ),%edx`	$10+4y$
`leal 7(%eax %ecx,2 ),%edx`	$9+x+2y$

练习题 3.4 答案

这个练习使你有机会检验你对操作数和算术指令的理解。

指令	目的	值
`addl %ecx,(%eax)`	0x100	0x100
`subl %edx,4(%eazx)`	0x104	0xA8
`imull $16,(eax,%edx,4)`	0x10C	0x110
`incl 8(eax)`	0x108	0x14
`decl %ecx`	%ecx	0x0
`subl %edx,%eax`	%eax	0xFD

练习题 3.5 答案

这个练习使你有机会生成一点汇编代码。答案的代码是由 GCC 生成的。将参数 n 加载到寄存器%ecx 中，它可以用字节寄存器%cl 来指定 sarl 指令的移位量。

```
1      movl 12(%ebp),%ecx   Get n
2      movl 8(%ebp),%eax    Get x
3      sall $2,%eax         x <<= 2
4      sarl %cl,%eax        x >>= n
```

练习题 3.6 答案

这个指令用来将寄存器%edx 设置为 0，运用了对任意 x，x ^ x = 0 这一属性。它对应于 C 语句 i = 0。

这是汇编语言习惯用法的一个示例，习惯用法就是常常用来完成特殊目的一段代码。认识这些习惯用法，是成为阅读汇编代码能手的第一步。

练习题 3.7 答案

这个例子要求你思考不同的比较和 set 指令。要注意的主要问题是，如果将比较指令一边的值强制类型转换成了 unsigned，那么由于隐式的强制类型转换，比较指令的执行就好像两边都是无符号数一样。

```
1    char ctest(int a, int b, int c)
2    {
3      char t1 =            a <              b;
4      char t2 =            b <   (unsigned) a;
5      char t3 = (short)    c >=  (short)  a;
6      char t4 = (char)     a !=  (char)   c;
7      char t5 =            c >              b;
8      char t6 =            a >              0;
9      return t1 + t2 + t3 + t4 + t5 + t6;
10   }
```

练习题 3.8 答案

这个练习要求你仔细检查反汇编代码，并推理出跳转目标的编码。它还使你练习了十六进制算术。

A. jbe 指令的目标为 0x8048d1c + 0xda。如原始反汇编代码所示，这就是 0x8048cf8。

```
8048d1c: 76 da              jbe     8048cf8
8048d1e: eb 24              jmp     8048d44
```

B. 根据反汇编器产生的注释，跳转目标是在绝对地址 0x8048d44。根据字节编码，这必须是在超过 mov 指令地址 0x54 字节地址的地方。减去 0x54 就得到 0x8048cf0，反汇编代码也证实了这一点：

```
8048cee: eb 54              jmp     8048d44
8048cf0: c7 45 f8 10 00     mov    $0x10,0xfffffff8(%ebp)
```

C. 目标是在相对于 0x8048907（nop 指令的地址）偏移量为 000000cb 的地方。对它们求和就得到地址 0x80489d2。

```
8048902: e9 cb 00 00 00      jmp      80489d2
8048907: 90                  nop
```

D. 间接跳转是用指令代码 ff 25 表示的。将要被读出的跳转目标的地址是由下面 4 个字节明确编码的。因为机器是小端法的，所以以反向顺序给出就是 e0 a2 04 08。

```
80483f0: ff 25 e0 a2 04      jmp      *0x804a2e0
80483f5: 08
```

练习题 3.9 答案

对汇编代码写注释，以及模仿它的控制流来编写 C 代码，是理解汇编语言程序很好的手段。本题使你能够练习一个具有简单控制流的示例。它还给你了一个检查逻辑操作实现的机会。

A.

code/asm/simple-if.c

```
1   void cond(int a, int *p)
2   {
3     if (p == 0)
4       goto done;
5     if (a <= 0)
6       goto done;
7     *p += a;
8    done:
9   }
```

code/asm/simple-if.c

B. 第一个条件分支是&&表达式实现的一部分。如果对 p 为非空的测试失败，代码会跳过对 a>0 的测试。

练习题 3.10 答案

编译循环产生的代码可能会难以分析，因为编译器会对循环代码进行很多不同的优化，还因为程序变量与寄存器的匹配非常困难。我们从非常简单的循环开始练习这种技能。

A. 只要看看是如何取出参数的，就能确定寄存器的使用。

寄存器用法		
寄存器	变量	初始
%esi	x	x
%ebx	y	y
%ecx	n	n

B. body-statement 部分是由 C 代码中的第 4~6 行和汇编代码中的第 6~8 行组成的。test-expr 部分是 C 代码中的第 7 行。在汇编代码中，它是由第 9~14 行的指令以及第 15 行的分支条件组成的。

C. 加了注释的代码是这样的：

Initially x, y, and n are at offsets 8, 12, and 16 from %ebp

```
1      movl 8(%ebp),%esi       Put x in %esi
2      movl 12(%ebp),%ebx      Put y in %ebx
3      movl 16(%ebp),%ecx      Put n in %ecx
4      .p2align 4,,7
5     .L6:                  loop:
6      imull %ecx,%ebx         y *= n
7      addl %ecx,%esi          x += n
8      decl %ecx n--
9      testl %ecx,%ecx         Test n
10     setg %al                n > 0
11     cmpl %ecx,%ebx          Compare y:n
12     setl %dl                y < n
13     andl %edx,%eax          (n > 0) & (y < n)
14     testb $1,%al            Test least significant bit
15     jne .L6                 If != 0, goto loop
```

注意，测试表达式的实现有点奇怪。很明显，编译器认出两个判定（n > 0）和（y < n）只可能取值 0 或 1，因此分支条件只需测试它们 AND 的最低字节。编译器还可以更聪明一点，用 testb 指令来执行 AND 操作。

练习题 3.11 答案

这个问题提供了另外一种机会来练习解读循环代码。C 编译器做了一些有趣的优化。

A. 看看参数是如何取出的，以及寄存器是如何初始化的，就能确定寄存器的使用。

寄存器用法		
寄存器	**变量**	**初始**
%eax	a	a
%ebx	b	b
%ecx	i	0
%edx	result	a

B. test-expr 出现在 C 代码的第 5 行，汇编代码的第 10 行，以及第 11 行的跳转条件。body-statement 出现在 C 代码的第 6～8 行，汇编代码的第 7～9 行。编译器发现 while 循环的初始测试总是为真的，因为 i 被初始化为 0，很明显小于 256。

C. 加了注释的代码是这样的：

```
1      movl 8(%ebp),%eax       Put a in %eax
2      movl 12(%ebp),%ebx      Put b in %ebx
3      xorl %ecx,%ecx          i = 0
4      movl %eax,%edx          result = a
```

```
5       .p2align 4,,7
     a in %eax, b in %ebx, i in %ecx, result in %edx
6     .L5:                         loop:
7       addl %eax,%edx             result += a
8       subl %ebx,%eax             a -= b
9       addl %ebx,%ecx             i += b
10      cmpl $255,%ecx             Compare i:255
11      jle .L5                    If <= goto loop
12      movl %edx,%eax             Set result as return value
```

D. 等价的 goto 代码是：

```
1   int loop_while_goto(int a, int b)
2   {
3     int i = 0;
4     int result = a;
5    loop:
6     result += a;
7     a -= b;
8     i += b;
9     if (i <= 255)
10      goto loop;
11    return result;
12  }
```

练习题 3.12 答案

一种分析汇编代码的方法是试着逆转编译过程，生成对 C 程序员来说看起来比较“自然的”C 代码。例如，我们不想使用 goto 语句，因为它在 C 中很少使用。很有可能我们也不使用 do-while 语句。这个练习迫使你将编译逆转成某种框架。它要求思考 for 循环的翻译。它还展示了一种称为**代码移动**（code motion）的优化技术，也就是当可以确定计算结果在循环中不会改变时，将计算从循环中拿出来。

A. 我们可以看出 result 必须在寄存器%eax 中。初始化时它被置为 0，循环结束时留在%eax 中作为返回值。我们可以看到 i 保存在寄存器%edx 中，因为这个寄存器是作为两个条件测试的基础的。

B. 第 2～4 行的指令将%edx 设置成 n-1。

C. 第 5 行和第 12 行的测试要求 i 非负。

D. 变量 i 被指令 4 减小。

E. 指令 1、6 和 7 使得 x*y 存储在寄存器%ecx 中。

F. 下面是原始代码：

```
1  int loop(int x, int y, int n)
2  {
3     int result = 0;
4     int i;
5     for (i = n-1; i >= 0; i = i-x) {
6        result += y * x;
7     }
8     return result;
9  }
```

练习题 3.13 答案

这个练习让你能够推算出开关语句的控制流。回答这些问题要求你将汇编代码中的多处信息综合起来：

1. 汇编代码的第 2 行将 x 加上 2，以将情况（cases）的下界设置成 0。这就意味着最小的情况标号（case lable）为-2。

2. 当调整过的情况值大于 6 时，第 3 行和第 4 行会导致程序跳转到默认情况。这就意味着最大情况标号为-2+6=4。

3. 在跳转表中，我们看到第二个表项（情况标号-1）的目的（.L10）与第 4 行的跳转指令的目的一样，表明这是默认情况行为。因此，在开关语句体中缺失了情况标号-1。

4. 在跳转表中，我们看到第 5 和第 6 个表项的目的一样。这对应于情况标号 2 和 3。

从上述推理，我们得到两个结论：

A. 开关语句体中的情况标号值为-2、0、1、2、3 和 4。

B. 目标为.L8 的情况标号为 2 和 3。

练习题 3.14 答案

这又是一个汇编代码的习惯用法。刚开始，它看起来非常奇怪——call 指令没有与之匹配的 ret。然后我们就意识到它根本就不是一个真正的过程调用。

A. %eax 被设置成 popl 指令的地址。

B. 这不是一个真正的子过程调用，因为控制是按照与指令相同的顺序进行的，而返回值是从栈中弹出的。

C. 这是 IA32 中将程序计数器的值放到整数寄存器中的惟一方法。

练习题 3.15 答案

这个练习使得对寄存器使用规则的讨论具体化。寄存器%edi、%esi 和%ebx 是被调用者保存的。在改变它们的值之前，过程必须将它们保存在栈中，在返回之前，要恢复它们。其他三个寄存器是调用者保存的，改变它们不会影响调用者的行为。

练习题 3.16 答案

能够推断函数是如何使用栈的，是理解编译器产生的代码的关键的一部分。正如这个例子说明

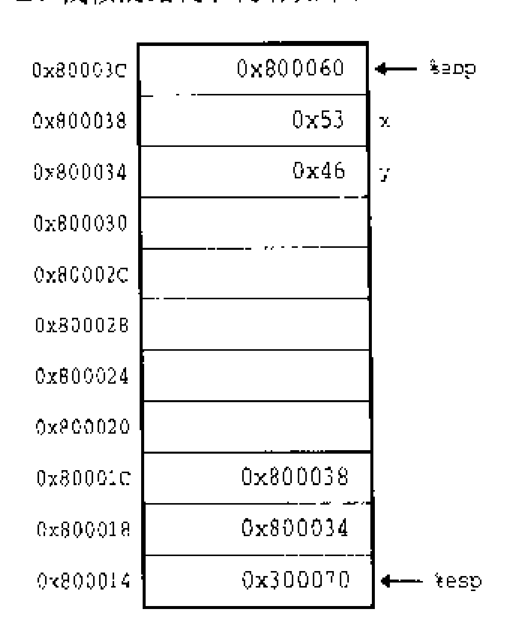
的那样，编译器分配了大量根本不会使用的空间。

A. 开始时，%esp 的值为 0x800040。第 2 行将这个值减了 4，得到 0x80003C，这就成为了%ebp 的新值。

B. 我们可以看到两个 leal 指令是如何计算要传给 scanf 的参数的。因为参数要以相反的顺序压入栈中，我们可以看到 x 位于相对于%ebp 偏移量为-4 的地方，而 y 在偏移量为-8 的地方。因此它们的地址是 0x800038 和 0x800034。

C. 栈指针的初始值为 0x800040，第 2 行将它减了 4。第 4 行将它减了 24，而第 5 行将它减了 4。三个入栈指令将它减了 12，总共减小了 44。因此，在第 10 行后，%esp 等于 0x800014。

D. 栈帧的结构和内容如下：



E. 0x800020～0x800033 的字节地址没有使用。

练习题 3.17 答案

这个练习测试你对数据大小和数组索引的理解。注意，任何类型的指针都是 4 个字节长。long double 的 GCC 实现用了 12 个字节来存储每个值，即使实际格式只需要 10 个字节。

数组	元素大小	总大小	起始地址	元素 i
S	2	28	x_S	x_S+2i
T	4	12	x_T	x_T+4i
U	4	24	x_U	x_U+4i
V	12	96	x_V	x_V+12i
W	4	16	x_W	x_W+4i

练习题 3.18 答案

这个练习是关于整数数组 E 的练习的一个变形。理解指针与指针指向的对象之间的区别是很重要的。因为数据类型 short 需要 2 个字节，所以所有的数组索引都将乘以因子 2。前面我们用的是

movl，现在用的则是 movw。

表达式	类型	值	汇编语句
`S+1`	`short *`	x_S+2	`leal 2(%edx),%eax`
`S[3]`	`short`	$M[x_S+6]$	`movw 6(%edx),%ax`
`&S[i]`	`short *`	x_S+2i	`leal (%edx, %ecx,2),%eax`
`S[4*i+1]`	`short`	$M[x_S+8i+2]$	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	$x_S+2i-10$	`leal -10(%edx,%ecx,2)%eaxs`

练习题 3.19 答案

这个练习要求你完成缩放指令，来确定地址的计算，并且应用行优先索引的公式。第一步是注释汇编代码，来确定如何计算地址引用：

```
1    movl 8(%ebp),%ecx             Get i
2    movl 12(%ebp),%eax            Get j
3    leal 0(,%eax,4),%ebx          4*j
4    leal 0(,%ecx,8),%edx          8*i
5    subl %ecx,%edx                7*i
6    addl %ebx,%eax                5*j
7    sall $2,%eax                  20*j
8    movl mat2(%eax,%ecx,4),%eax   mat2[(20*j + 4*i)/4]
9    addl mat1(%ebx,%edx,4),%eax   + mat1[(4*j + 28*i)/4]
```

由此我们可以看出，对矩阵 mat1 的引用是在字节偏移 4(7i+j)的地方，而对矩阵 mat2 的引用是在字节偏移 4(5j+i)的地方。由此我们可以确定 mat1 有 7 列，而 mat2 有 5 列，得到 M=5 和 N=7。

练习题 3.20 答案

这个练习要求你研究汇编代码，理解是如何优化它的。对提高程序性能来说，这是一项很重要的技能。通过调整你的源代码，你可以影响产生的机器代码的效率。

下面是该 C 代码的一个优化过的版本：

```
1  /* Set all diagonal elements to val */
2  void fix_set_diag_opt(fix_matrix A, int val)
3  {
4    int *Aptr = &A[0][0] + 255;
5    int cnt = N-1;
6    do {
7       *Aptr = val;
8       Aptr -= (N+1);
9       cnt--;
```

```
10    } while (cnt >= 0);
11 }
```

通过下面的注释可以看出它与汇编代码的关系：

```
1      movl 12(%ebp),%edx       Get val
2      movl 8(%ebp),%eax        Get A
3      movl $15,%ecx            i = 0
4      addl $1020,%eax          Aptr = &A[0][0] + 1020/4
5      .p2align 4,,7
6    .L50:                    loop:
7      movl %edx,(%eax)         *Aptr = val
8      addl $-68,%eax           Aptr -= 68/4
9      decl %ecx                i--
10     jns .L50                 if i >= 0 goto loop
```

请注意汇编代码是如何从数组结尾处开始并反向工作的。它将指针减去68（=17·4），因为数组元素A[i-1][i-1]和A[i][i]之间隔着N+1个元素。

练习题3.21 答案

这个练习让你思考结构布局，以及用来访问结构的域的代码。该结构声明是文中所示例子的一个变形。它表明嵌套的结构的分配是将内层结构嵌入到外层结构之中的。

A 结构的布局图是这样的：

偏移	0	4	8	12
内容	p	s.x	s.y	next

B. 它使用了16个字节。

C 同平时一样，我们从给汇编代码加注释开始：

```
1      movl 8(%ebp),%eax        Get sp
2      movl 8(%eax),%edx        Get sp->s.y
3      movl %edx,4(%eax)        Copy to sp->s.x
4      leal 4(%eax),%edx        Get &(sp->s.x)
5      movl %edx,(%eax)         Copy to sp->p
6      movl %eax,12(%eax)       sp->next = p
```

由此，我们可以产生如下C代码：

```
void sp_init(struct prob *sp)
{
    sp->s.x   = sp->s.y;
    sp->p     = &(sp->s.x);
```

```
    sp->next = sp;
}
```

练习题 3.22 答案

这是一个很棘手的问题。它将对以猜谜技术作为逆向工程的一部分的需求提升到了一个新高度。它清晰地表明，联合是一种将多个名字（和类型）与单个存储位置联系到一起的简单方法。

A. 联合的布局如下面所示。正如这张表说明的那样，这个联合既可以解释为“e1”（有域 e1.p 和 e1.y），也可以解释为“e2”（有域 e2.x 和 e2.next）。

偏移	0	4
内容	e1.p	e1.y
	e2.x	e2.next

B. 它使用了 8 个字节。

C. 同平时一样，我们从给汇编代码加注释开始。在我们的注释中，对有些指令，我们给出了多个可能的解释，同时还指出后面会丢弃哪一种解释。例如，第 2 行既可以解释成获取元素 e1.y，也可以解释成获取元素 e2.next。在第 3 行，我们看到是用间接存储器引用的方式来使用这个值的，所以只可能是第二种解释了。

```
1    movl 8(%ebp),%eax     Get up
2    movl 4(%eax),%edx     up->e1.y (no) or up->e2.next
3    movl (%edx),%ecx      up->e2.next->e1.p or up->e2.next->e2.x (no)
4    movl (%eax),%eax      up->e1.p (no) or up->e2.x
5    movl (%ecx),%ecx      *(up->e2.next->e1.p)
6    subl %eax,%ecx        *(up->e2.next->e1.p) - up->e2.x
7    movl %ecx,4(%edx)     Store in up->e2.next->e1.y
```

由此，我们可以产生如下 C 代码：

```
void proc (union ele *up)
{
    up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
}
```

练习题 3.23 答案

对理解各种数据结构需要多少存储，以及对理解编译器为访问这些结构产生的代码来说，理解结构布局和对齐是非常重要的。这个练习让你看清楚一些示例结构的细节。

A. `struct P1 { int i; char c; char d; int j; };`

i	c	j	d	总共	对齐
0	4	8	12	16	4

B. `struct P2 { int i; char c; char d; int j; };`

i	c	j	d	总共	对齐
0	4	5	8	12	4

C. `struct P3 { short w[3]; char c[3] };`

w	c	总共	对齐
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

w	c	总共	对齐
0	8	20	4

E. `struct P3 { struct P1 a[2]; struct P2 *P };`

a	p	总共	对齐
0	32	36	4

练习题 3.24 答案

这个问题覆盖的话题比较广泛，例如栈帧、字符串表示、ASCII 码和字节顺序。它说明了越界存储器引用的危险性，以及缓冲区溢出背后的基本思想。

A. 第 7 行时的栈。

08	04	86	43	返回地址
bf	ff	fc	94	保存的 %ebp ← %ebp
				buf[4-7]
				buf[0-3]
00	00	00	01	保存的 %esi
00	00	00	02	保存的 %ebx

B. 第 10 行后的栈（只给出了修改过的字）。

08	04	86	00	返回地址
31	30	39	38	保存的 %ebp ← %ebp
37	36	35	34	buf[4-7]
33	32	31	30	buf[0-3]

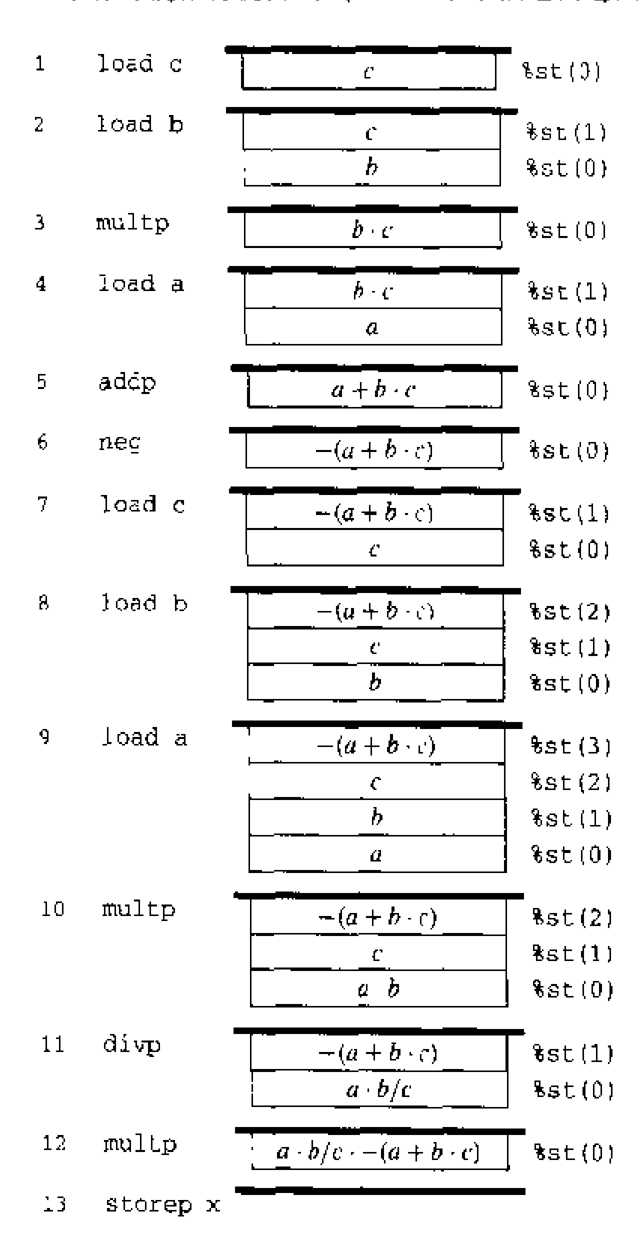
C. 这个程序试图返回到地址 0x08048600，低位字节被结尾的空（null）字符覆盖了。

D. 保存的寄存器%ebp 的值变成了 0x31303938，会在 getline 返回之前，将这个值加载到寄存器中。保存的其他寄存器不受影响，因为它们保存在栈中比 buf 更低的地址上。

E. 对 malloc 的调用应该以 strlen(buf)+1 作为它的参数，而且还应该检查返回值是否为非空。

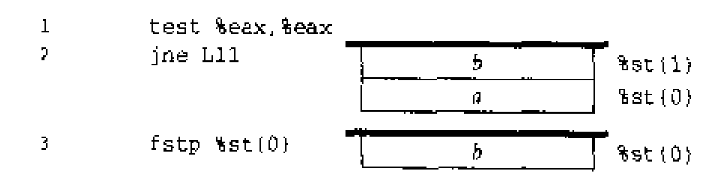
练习题 3.25 答案

这个练习使你有机会试试 3.14.2 节中描述的递归过程。



练习题 3.26 答案

下面这段代码与编译器为基于一个测试结果从两个值中进行选择产生的代码相似：

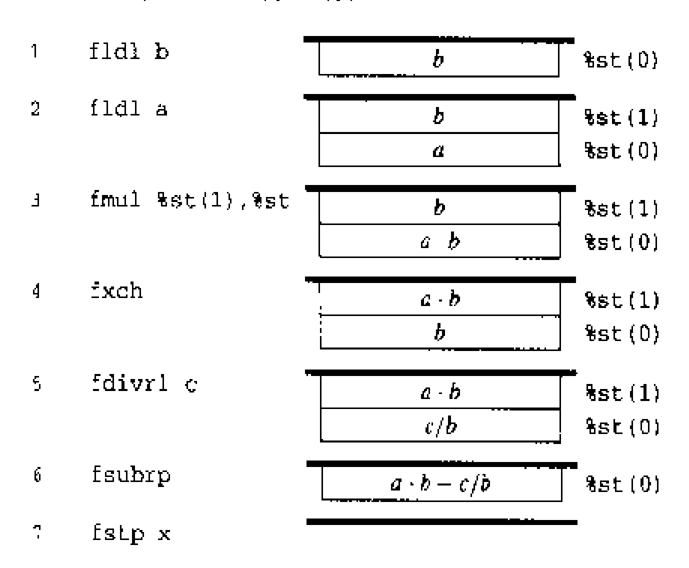


```
4       jmp L9
5   L11:
6       fstp %st(1)        %st(0): a
7   L9:
```

得到的栈顶值是 x ? a : b。

练习题 3.27 答案

由于它的关于弹出操作数的规则，以及参数的顺序等等，浮点代码非常难处理。这个练习使你有机会完整地完成一些特殊情况。



这段代码计算的是表达式 x=a*b-c/b。

练习题 3.28 答案

这个练习要求你考虑浮点代码中的各种操作数的类型和大小。

code/asm/fpfunct2-ans.c

```
1  double funct2(int a, double x, float b, float i)
2  {
3      return a/(x+b) - (i+1);
4  }
```

code/asm/fpfunct2-ans.c

练习题 3.29 答案

在第 4 行和第 5 行之间插入下列代码：

```
1       cmpb $1,%ah       Test if comparison outcome is <
```

练习题 3.30 答案

```
1  int ok_smul(int x, int y, int *dest)
2  {
```

```
3       long long prod = (long long) x * y;
4       int trunc = (int) prod;
5
6       *dest = trunc;
7       return (trunc == prod);
8 }
```

CHAPTER 4

处理器体系结构

现代微处理器可以称得上是人类创造出的最复杂的系统之一。一块手指甲大小的硅片上，可以容纳一个完整的高性能处理器和大的高速缓存，以及用来连接外部设备的逻辑电路。从性能上来说，今天在一块芯片上实现的处理器已经使20年前价值1000万美元、有房间那么大的超级计算机相形见绌了。即使是在像手机、个人数字助理和掌上游戏机这样的日常设备中的嵌入式处理器，也比早期计算机开发者所能想到的强大得多。

到目前为止，我们看到的计算机系统只限于机器语言程序级。我们知道处理器必须执行一系列指令，每条指令执行某个简单操作，例如两个数相加。指令被编码为由一个或多个字节序列组成的二进制格式。一个处理器支持的指令和指令的字节级编码称为它的ISA（instruction-set architecture，指令集体系结构）。不同的处理器"家族"，例如Intel IA32、IBM/Motorola PowerPC和Sun Microsystems SPARC，都有不同的ISA。一个程序编译成在一种机器上运行，就不能在另一种机器上运行。另外，同一个家族里也有很多不同类型的处理器。虽然每个厂商制造的处理器性能和复杂性不断提高，但是不同的类型在ISA级别上都保持着兼容。一些常见的处理器家族（例如IA32）中的处理器分别由多个厂商提供。因此，ISA在编译器编写者和处理器设计人员之间提供了一个概念抽象层，编译器编写者只需要知道允许哪些指令，以及它们是如何编码的；而处理器设计者必须建造出执行这些指令的处理器。

本章将简要介绍处理器硬件的设计。我们将研究一个硬件系统执行某种ISA指令的方式，这会使你能更好地理解计算机是如何工作的，以及计算机制造商们面临的技术挑战。一个很重要的概念就是现代处理器的实际工作方式可能跟ISA隐含的计算模型大相径庭。ISA模型看上去应该是顺序指令执行，也就是先取出一条指令，等到它执行完毕，再开始下一条。然而，与一个时刻只执行一条指令相比，通过同时处理多条指令的不同部分，处理器可以获得较高的性能。为了保证处理器能得到同顺序执行相同的结果，人们采用了一些特殊的机制。在计算机科学中，用巧妙的方法在提高性能的同时，又保持一个更简单、更抽象模型的功能的思想是众所周知的。在Web浏览器或像平衡二叉树和哈希表这样的信息检索数据结构中使用缓存，就是这样的例子。

你很可能永远都不会自己设计处理器。这是专家们的任务，他们工作在全球不到100家的公司里。那么为什么你还应该了解处理器设计呢？

- 从智力方面来说，处理器设计是非常有趣的。学习处理器是怎样工作的本身就是一件很有意义的事情。而格外有趣的事情是了解作为计算机科学家和工程师日常生活一部分的一个系统的内部工作原理，特别是很多人都还不了解它。处理器设计包括许多好的工程实践原理。它需要完成复杂的任务，而结构又要尽可能简单。
- 理解处理器是如何工作的能帮助理解整个计算机系统是如何工作的。在第6章中，我们将讲述存储器（memory）系统以及用来创建很大的存储器映像同时又有快速访问时间的技术。参考处理器端的处理器—存储器接口会使那些讲述更完整。
- 虽然很少有人设计处理器，但是许多人设计包含处理器的硬件系统。将处理器嵌入到实际系统中，如汽车和家用电器，已经变得非常普通了。嵌入式系统的设计者必须了解处理器是如何工作的，因为这些系统通常是在比桌面系统更低抽象级别上进行设计和编程的。
- 你的工作可能就是处理器设计。虽然生产处理器的公司很少，但是研究处理器的设计人员队伍已经非常巨大了，而且还在增大。一个主要的处理器设计的各个方面大约涉及到800多人。

本章中，我们首先要定义一个简单的指令集，用来作为我们处理器实现的运行示例。因为受到IA32指令集的启发，而它又被称为"X86"，所以我们称我们的指令集为"Y86"指令集。与IA32相比，Y86指令集的数据类型、指令和寻址方式都要少一些，它的字节级编码也比较简单。不过，它仍然足够完整，能让我们写一些简单的处理整数的程序。设计一个实现Y86的处理器要求我们面对许多处理器设计者同样会面对的问题。

接下来我们会提供一些数字硬件设计的背景。我们会描述处理器中使用的基本构件块，以及它们是如何连接起来和操作的。这些介绍是建立在第2章对布尔代数和位操作的讨论的基础上的。我们还将介绍一种描述硬件系统控制部分的简单语言，HCL（Hardware Control Language，硬件控制语言）。过后，我们会用它来描述我们的处理器设计。即使你已经有了一些逻辑设计的背景知识，也应该读读这个部分以了解我们的特殊符号。

作为设计处理器的第一步，我们给出一个基于顺序操作、功能正确但是有点不实用的Y86处理器。这个处理器每个时钟周期执行一条完整的Y86指令。所以它的时钟必须足够慢，以允许在一个周期内完成所有的动作。这样一个处理器是可以实现的，但是它的性能远远低于相同硬件应该能达到的性能。

以这个顺序设计为基础，我们进行一些改造，创建一个流水线化的处理器（pipelined processor）。这个处理器将每条指令的执行分解成五步，每个步骤由一个独立的硬件部分或阶段（stage）来处理。指令步经流水线的各个阶段，且每个时钟周期有一条新指令进入流水线。所以，处理器可以同时执行五条指令的不同阶段。为了使这个处理器保留Y86 ISA的顺序的性质，就要求处理很多冒险或冲突（hazard）条件。冒险就是一条指令的位置或操作数依赖于其他仍在流水线中的指令。

我们设计了一些工具来研究和测试我们的处理器设计。其中包括Y86的编译器、在你的机器上运行Y86程序的模拟器，还有针对两个顺序处理器设计和一个流水线化处理器设计的模拟器。这些设计的控制逻辑是在用HCL符号表示的文件中描述的。通过编辑这些文件和重新编译模拟器，你可以改变和扩展模拟行为。我们还提供许多练习，包括实现新的指令和修改机器处理指令的方式，还提供测试代码以帮助你评价你修改的正确性。这些练习将极大地帮助你理解所有这些内容，也能使你更理解处理器设计者面临的许多不同的设计选择。

4.1 Y86指令集体系结构

如图4.1所示，Y86程序中的每条指令都会读取或修改处理器状态的某些部分。这称为程序员可见状态，这里的"程序员"既指用汇编代码写程序的人，也包括产生机器级代码的编译器。在我们的处理器实现中，只要我们能保证机器级程序能够访问程序员可见状态，就不需要完全按照ISA隐含的方式来表示和组织这个处理器状态。Y86的处理器状态类似于IA32。有八个程序寄存器：%eax、%ecx、%edx、%ebx、%esi、%edi、%esp和%ebp，处理器每个程序寄存器存储一个字。寄存器%esp被入栈、出栈、调用和返回指令作为栈指针。而其他寄存器没有固定的含义或固定值。有三个一位的条件码：ZF、SF和OF，它们保存着有关最近的算术或逻辑指令造成影响的信息。程序计数器（PC）里存放着当前正在执行指令的地址。存储器，从概念上来说就是一个很大的字节数组，保存着程序和数据。Y86程序用虚拟地址来引用存储器位置。硬件和操作系统软件联合起来将虚拟地址翻译成指明数据实际存在存储器中哪个地方的实际或物理地址。我们还将在第10章中进一步详

细讨论虚拟存储器。现在，我们只认为虚拟存储器提供给 Y86 程序一个统一的字节数组映像。

程序寄存器

%eax	%esi
%ecx	%edi
%edx	%esp
%ebx	%ebp

条件码

ZF SF OF

PC

存储器

图 4.1 Y86 程序员可见状态

同 IA32 一样，Y86 的程序可以访问和修改程序寄存器、条件码、程序计数器（PC）和存储器。

图 4.2 给出了 Y86 ISA 中各个指令的简单描述。这个指令集就是我们处理器实现的目标。Y86 指令集基本上是 IA32 指令集的一个子集。它只包括四字节整数操作，寻址方式比较少，操作也较少。因为我们只有四字节数据，所以称之为“字（word）”。在这个图中，左边是指令的汇编码表示，右边是字节编码。汇编码和 IA32 程序的 GAS 表示非常类似。

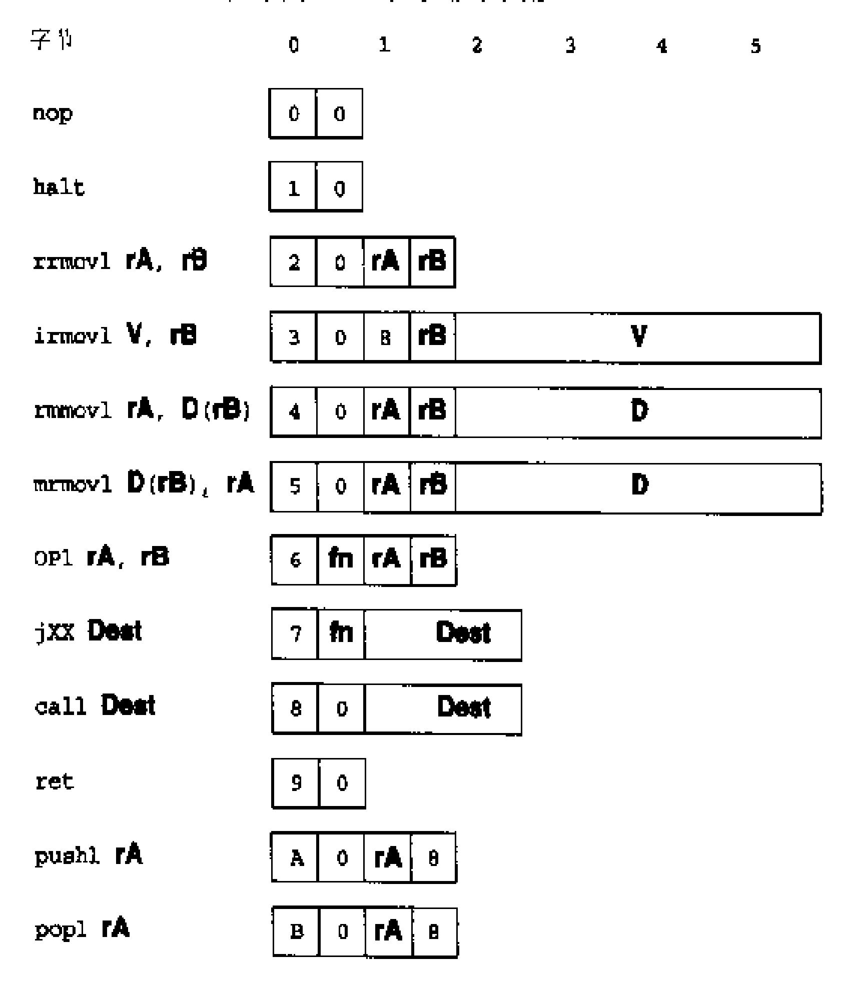


图 4.2 Y86 指令集

指令编码从 1 个字节到 6 个字节不等。一条指令含有一个单字节的指令指示符，可能含有一个单字节的寄存器指示符，还可能含有一个四字节的常数字。字段 fn 指明是某个整数操作（OPl）或是某个分支条件（jXX）。所有的数值都用十六进制表示。

下面是不同 Y86 指令的更多细节。

- IA32 的 movl 指令分成了四个不同的指令：irmovl、rrmovl、mrmovl 和 rmmovl，分别显式地指明源和目的的格式。源可以是立即数（i）、寄存器（r）或存储器（m）。指令名字的第一个字母就表明了源的类型。目的可以是寄存器（r）或存储器（m）。指令名字的第二个字母指明了目的的类型。在决定如何实现它们时，显式地指明数据传送的这四种类型是很有帮助的。

  两个存储器传送指令中的存储器引用方式是简单的基址加位移形式。在地址计算中，我们不支持第二变址寄存器（second index register）和任何寄存器值的伸缩（scaling）。

  同 IA32 一样，我们不允许从一个存储器地址直接传送到另一个存储器地址。另外，我们也不允许将立即数传送到存储器。
- 有四个整数操作指令，就是图 4.2 中的 OPl。它们是 addl、subl、andl 和 xorl。它们只对寄存器数据进行操作，而 IA32 还允许对存储器数据进行这些操作。这些指令会设置三个条件码 ZF、SF 和 OF（零、符号和溢出）。
- 七个跳转指令（图 4.2 中的 jxx）是 jmp、jle、jl、je、jne、jge 和 jg。根据分支指令的类型和条件代码的设置来选择分支。分支条件和 IA32 的一样（见图 3.11）。
- call 指令将返回地址入栈，然后跳到目的地址。ret 指令从这样的过程调用中返回。
- pushl 和 popl 指令实现了入栈和出栈，就像在 IA32 中一样。
- halt 指令停止指令的执行。IA32 中有一个与之相当的指令，叫 hlt。IA32 的应用程序不允许使用这条指令，因为它会导致整个系统停止。我们在 Y86 程序中用 halt 指令来停止模拟器。

图 4.2 还给出了指令的字节级编码，取决于需要那些字段，每条指令需要 1～6 个字节不等。每条指令的第一个字节表明指令的类型。这个字节分为两个部分，每部分四位：高四位是代码（code）部分，低四位是功能（function）部分。如图 4.2 所示，代码值为 0～B（十六进制）。功能值只有在一组相关指令共用一个代码时才有用。图 4.3 给出了整数操作和分支指令的具体编码。

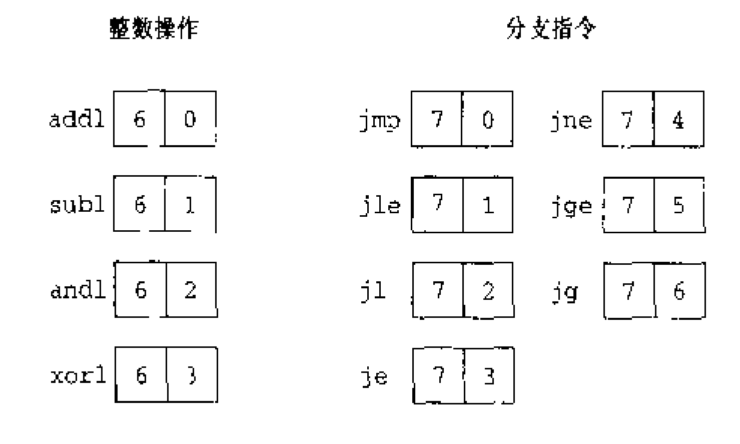


图 4.3 Y86 指令集的功能码

这些代码指明是某个整数操作还是分支条件。这些指令是图 4.2 中所示的 OPl 和 jXX。

如图 4.4 所示，八个程序寄存器中每个都有相应的 0～7 的寄存器标识符（register ID）。Y86 中的寄存器编号跟 IA32 中的相同。程序寄存器被存在 CPU 中的一个寄存器文件中，这个寄存器文件

就是一个小的、以寄存器ID作为地址的随机访问存储器。ID值8用于指令编码中，在我们的硬件设计中，当需要指明不应访问任何寄存器时，我们就用这个值来表示。

数字	寄存器名字
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
8	无寄存器

图 4.4 Y86程序寄存器标识符

八个程序寄存器中每个都有一个相对应的标识符（ID），0~7。如果指令中某个寄存器字段的值为ID 8，就表明此处没有寄存器操作数。

有的指令只有一个字节长，而有的需要操作数的指令编码就更长一些。首先，可能有附加的寄存器指示符字节（register specifier byte），指定一个或两个寄存器。在图4.2中，这些寄存器字段称为rA和rB。从指令的汇编代码表示中可以看到，根据指令类型，指令可以指定用于数据源和目的的寄存器，或是用于地址计算的基址寄存器。没有寄存器操作数的指令，例如分支指令和调用指令，就没有寄存器指示符字节。那些只需要一个寄存器操作数的指令（irmovl、pushl和popl）将另一个寄存器指示符设为8。这种约定在我们的处理器实现中非常有用。

有些指令需要一个附加的四字节常数字（constant word）。这个字能作为irmovl的立即数数据，作为rmmovl和mrmovl的地址指示符的位移量，以及分支指令和调用指令的目的地址。注意，分支指令和调用指令的目的是一个绝对地址，而不像IA32中那样使用PC（程序计数器）相关的寻址方式。处理器使用PC相关的寻址方式，分支指令的编码会更简洁，同时这样也能允许代码从存储器的一部分拷贝到另一部分而不需要更新所有的分支目标地址。因为我们更关心描述的简单性，所以就使用了绝对寻址方式。同IA32一样，所有整数采用小端法（little-endian）编码。当指令按照反汇编格式书写时，这些字节就以相反的顺序出现。

例如，让我们用十六进制来表示指令 `rmmovl %esp, 0x12345(%edx)` 的字节编码。从图4.2我们可以看到，rmmovl的第一个字节为40。源寄存器%esp应该编码放在rA字段中，而基址寄存器%edx应该编码放在rB字段中。根据图4.4中的寄存器编号，我们得到寄存器指示符字节42。最后，位移量编码放在四字节的常数字中。首先在0x12345的前面填充上0变成4个字节，变成字节序列00 01 23 45。写成按字节反序就是45 23 01 00。将它们都连接起来就得到指令的编码404245230100。

指令集的一个重要性质就是字节编码必须有惟一的解释。任意一个字节序列要么是一个惟一的指令序列的编码，要么就不是一个合法的字节序列。Y86就具有这个性质，因为每条指令的第一个字节有惟一的代码和功能组合，给定这个字节，我们就可以决定所有其他附加字节的长度和含义。这个性质保证了处理器可以无二义性地执行目标代码程序。只要从序列的第一个字节开始

处理，即使代码嵌入在程序中其他字节中，我们仍然可以很容易地确定指令序列。反过来说，如果不知道一段代码序列的起始位置，我们就不能准确地确定怎样将序列划分成单独的指令。对于试图直接从目标代码字节序列中抽取出机器级程序的反汇编程序和其他一些工具来说，这就带来了问题。

练习题 4.1

确定下面的 Y86 指令序列的字节编码。".pos 0x100" 那一行表明这段目标代码的起始地址应该是 0x100。

```
.pos 0x100   # Start generating code at address 0x100
    irmovl  $15,%ebx
    rrmovl  %ebx,%ecx
loop:
    rmmovl  %ecx,-3(%ebx)
    addl    %ebx,%ecx
    jmp loop
```

练习题 4.2

确定下面列出的每个字节序列代表的 Y86 指令序列。如果序列中有不合法的字节，指出指令序列中不合法值出现的位置。每个序列都先给出了起始地址，冒号后是字节序列。

```
A. 0x100:3083fcffffff40630008000010
B. 0x200:a068800080200001030830a00000090
C. 0x300:505407000000000f0b018
D. 0x400:6113730004000010
E. 0x500:6362a080
```

旁注：比较 IA32 和 Y86 的指令编码

同 IA32 中的指令编码相比，Y86 的编码简单得多，但是也没那么简洁。在所有的 Y86 指令中，寄存器字段的位置都是固定的，而在不同的 IA32 指令中，它们的位置是不一样的。即使最多只有 8 个寄存器，我们也对寄存器采用了 4 位编码。IA32 只用了 3 位编码。所以 IA32 能将入栈或出栈指令放在一个字节里，5 位字段表明指令类型，剩下的 3 位是寄存器指示符。IA32 可以将常数值编码成 1、2 或 4 个字节，而 Y86 总是将常数值编码成 4 个字节。

旁注：RISC 和 CISC 指令集

IA32 有时被称为"复杂指令集计算机"（CISC——读作"sisk"），与"精简指令集计算机"（RISC——读作"risk"）相对。从历史上看，从最早的计算机发展而来，先出现了 CISC 机器，到 20 世纪 80 年代早期，由于机器设计者加入了很多新指令来支持高级任务，例如，处理循环缓冲区，执行小数计算，以及求多项式的值，大型机和小型机的指令集已经变得非常庞大了。最早的微处理器出现在 20 世纪 70 年代早期，因为当时的集成电路技术极大地制约了一块芯片上能实现些什么，所以它们的指令集非常有限。微处理器发展得很快，到 20 世纪 80 年代以前，大型机和小型机的指令集复杂度一直都在增加，80x86 家族沿着这条道路，发展到了 IA32。即使是 IA32 也仍然在不断增加新

的指令类，来支持处理多媒体应用的需要。

RISC的设计理念是在20世纪80年代早期作为上述发展趋势的一种替代发展起来的。IBM的一组硬件和编译器专家受到IBM研究员John Cocke的很大影响，认为他们可以为更简单的指令集形式产生高效的代码。实际上，许多加到指令集中的高级指令很难被编译器产生，并且这些指令也很少被用到。一个较为简单的指令集可以用很少的硬件实现，能以高效的流水线结构组织起来，与本章后面描述的情况很类似。IBM直到多年以后才将这个理念商品化，开发出了Power和PowerPC ISA。

加州大学伯克利分校的David Patterson和斯坦福大学的John Hennessy进一步发展了RISC的概念。Patterson将这种新的机器类型命名为RISC，而将以前的那种称为CISC，因为以前没有必要给一种几乎是通用的指令集格式起名字。

比较CISC和最初的RISC指令集，我们发现下面这样一些一般特性。

CISC	早期的RISC
指令数量很多。Intel描述全套指令的文档[19]有700多页长	指令数量少得多。通常少于100个
有些指令的执行时间很长。包括将一个整块从存储器的一个部分拷贝到另一部分的指令，以及其他一些将多个寄存器的值拷贝到存储器或从存储器拷贝到多个寄存器的指令	没有较长执行时间的指令。有些早期的RISC机器甚至没有整数乘法指令，要求编译器通过一系列加法来实现乘法
编码是可变长度的。IA32的指令长度可以是1~15个字节	编码是固定长度的。通常所有的指令都编码为4个字节
指定操作数的方式很多样。在IA32中，存储器操作数指示符可以有许多不同的组合，这些组合由位移、基址和变址寄存器以及伸缩因子组成	简单寻址方式。通常只有基址和位移寻址
可以对存储器和寄存器操作数进行算术和逻辑运算	只能对寄存器操作数进行算术和逻辑运算。允许使用存储器引用的只有load和store指令。load是从存储器读到寄存器，store是从寄存器写到存储器。这种方法被称为load/store体系结构
对机器级程序来说实现细节是不可见的。ISA提供了程序和如何执行程序之间的清晰的抽象	对机器级程序来说实现细节是可见的。有些RISC机器禁止某些特殊的指令序列，而有些跳转要到下一条指令执行完了以后才会生效。编译器必须在这些约束条件下进行性能优化
条件码。作为指令执行的副产品，设置了一些特殊的标志位，可以用来作为条件分支检测	没有条件码。相反，对条件检测来说，要用明确的测试指令，这些指令会将测试结果放在一个普通的寄存器中
栈密集的过程链接。栈被用来存取过程参数和返回地址	寄存器密集的过程链接。寄存器被用来存取过程参数和返回地址。因此有些过程能完全避免存储器引用。通常处理器有更多的（最多的有32个）寄存器

Y86指令集既有CISC指令集的属性，也有RISC指令集的属性。和CISC一样，它有条件码、指令长度可变，以及栈密集的过程链接。和RISC一样的是，它采用load/store体系结构和规则编码（regular encoding）。Y86指令集可以看成是采用的CISC指令集（IA32），但又根据某些RISC的原理进行了简化。

旁注：RISC 与 CISC 之争

贯穿整个 20 世纪 80 年代，计算机体系结构领域里关于 RISC 指令集和 CISC 指令集优缺点的争论十分激烈。RISC 的支持者声称在给定硬件数量的情况下，通过结合简约式指令集设计、高级编译技术和流水线化的处理器实现，他们能够得到更强的计算能力。而 CISC 的拥趸反驳说要完成一个给定的任务只需要用较少的 CISC 指令，而且这样的机器能够获得较高的总体性能。

大多数公司都推出了 RISC 处理器产品，包括 Sun Microsystems（SPARC）、IBM 和 Motorola（PowerPC），以及 Digital Equipment Corporation（Alpha）。

在 20 世纪 90 年代早期，争论逐渐平息，因为事实已经很清楚了，无论是单纯的 RISC 还是单纯的 CISC 都不如结合两者思想精华的设计。RISC 机器发展进化的过程中，引入了更多的指令，而许多这样的指令都需要执行多个周期。今天的 RISC 机器的指令表中有几百条指令，几乎与"精简指令集机器"的名称不相符了。那种将实现细节暴露给机器级程序的思想已经被证明是短视的了。随着使用更加高级硬件结构的新处理器模型的开发，许多实现细节已经变得很落后了，但它们仍然是指令集的一部分。不过，作为 RISC 设计的核心的指令集仍然是非常适合在流水线化的机器上执行的。

比较新的 CISC 机器也利用了高性能流水线结构。就像我们将在 5.7 节中讨论的那样，它们读取 CISC 指令，并动态地翻译成比较简单的、像 RISC 那样的操作的序列。例如，一条将寄存器和存储器相加的指令被翻译成三个操作：一个是读原始的存储器值，一个是执行加法运算，第三就是将和写回存储器。由于动态翻译通常可以在实际指令执行前进行，处理器可以保持很高的执行率。

除了技术因素以外，市场因素也在决定不同指令集是否成功中起了很重要的作用。通过保持与现有处理器的兼容，Intel 以及 IA32 使得从一代处理器迁移到下一代变得很容易。由于集成电路技术的进步，Intel 和其他 IA32 处理器制造商能够克服原来 8086 指令集设计造成的低效率，使用 RISC 技术产生出与最好的 RISC 机器相当的性能。在桌面和便携计算领域里，IA32 占据了完全的统治地位。

RISC 处理器在嵌入式处理器市场上表现得非常出色。嵌入式处理器负责控制移动电话、汽车刹车以及因特网设备等系统。在这些应用中，降低成本和功耗比保持后向兼容性更重要。就出售的处理器数量来说，这是个非常广阔而迅速成长着的市场。

图 4.5 给出了下面这个 C 函数的 IA32 和 Y86 汇编代码：

```
int Sum(int *Start, int Count)
{
    int sum = 0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}
```

IA32代码

*int Sum(int *Start, int Count)*

```
1   Sum:
2     pushl %ebp
3     movl %esp,%ebp
4     movl 8(%ebp),%ecx        ecx  = Start
5     movl 12(%ebp),%edx       edx  = Count
6     xorl %eax,%eax           sum  = 0
7     testl %edx,%edx
8     je .L34
9   .L35:
10    addl (%ecx),%eax         add *Start to sum
11    addl $4,%ecx             Start++
12    decl %edx                Count--
13    jnz .L35                 Stop when 0
14  .L34:
15    movl %ebp,%esp
16    popl %ebp
17    ret
```

Y86 代码

*int Sum(int *Start, int Count)*

```
1   Sum:
2     pushl %ebp
3     rrmovl %esp,%ebp
4     mrmovl 8(%ebp),%ecx     ecx  = Start
5     mrmovl 12(%ebp),%edx    edx  = Count
6     xorl %eax,%eax          sum  = 0
7     andl %edx,%edx
8     je End
9   Loop:
10    mrmovl (%ecx),%esi      get *Start
11    addl %esi,%eax          add to sum
12    irmovl $4,%ebx
13    addl %ebx,%ecx          Start++
14    irmovl $-1,%ebx
15    addl %ebx,%edx          Count--
16    jne Loop                Stop when 0
17  End:
18    rrmovl %ebp,%esp
19    popl %ebp
20    ret
```

图 4.5 Y86 汇编程序与 IA32 汇编程序比较

Sum 函数计算一个整数数组的和。Y86 代码与 IA32 代码的主要区别在于，它可能需要多条指令来执行一条 IA32 指令所完成的功能。

这段IA32代码是C编译器GCC产生的。Y86代码实质上是一样的，除了Y86有时需要两条指令来完成IA32一条指令就能完成的事情。如果我们用数组索引来写这个程序，要转换成Y86代码就困难多了，因为Y86没有伸缩（scaled）寻址模式。

图4.6给出了一个用Y86汇编代码编写的一个完整的程序文件的例子。这个程序既包括数据，也包括指令。命令（directive）指明应该将代码或数据放在什么位置，以及如何对齐（align）。这个程序详细说明了栈的放置、数据初始化、程序初始化和程序结束等问题。

code/arch/y86-code/asum.ys

```
1   # Execution begins at address 0
2           .pos 0
3   init:   irmovl Stack, %esp      # Set up Stack pointer
4           irmovl Stack, %ebp      # Set up base pointer
5           jmp Main                # Execute main program
6
7   # Array of 4 elements
8           .align 4
9           array: .long 0xd
10                 .long 0xc0
11                 .long 0xb00
12                 .long 0xa000
13
14  Main:   irmovl $4,%eax
15          pushl %eax              # Push 4
16          irmovl array,%edx
17          pushl %edx              # Push array
18          call Sum                # Sum(array, 4)
19          halt
20
21          # int Sum(int *Start, int Count)
22  Sum:    pushl %ebp
23          rrmovl %esp,%ebp
24          mrmovl 8(%ebp),%ecx     # ecx = Start
25          mrmovl 12(%ebp),%edx    # edx = Count
26          irmovl $0, %eax         # sum = 0
27          andl %edx,%edx
28          je End
29  Loop:   mrmovl (%ecx),%esi      # get *Start
30          addl %esi,%eax          # add to sum
31          irmovl $4,%ebx          #
32          addl %ebx,%ecx          # Start++
33          irmovl $-1,%ebx         #
34          addl %ebx,%edx          # Count--
35          jne Loop                # Stop when 0
36  End:    popl %ebp
```

```
37              ret
38              .pos 0x100
39   Stack:     # The stack goes here
```

code/arch/y86-code/asum.ys

图 4.6 用 Y86 汇编代码写的一个例子程序

调用 Sum 函数来计算 4 元素数组的和。

在这个程序中，以“.”开头的词是汇编器命令（assembler directives），它们告诉汇编器调整地址，以便在那儿产生代码或插入一些数据。命令.pos 0（第 2 行）告诉汇编器应该从地址 0 处开始产生代码。这个地址是所有 Y86 程序的起点。接下来的两条指令（第 3 行和第 4 行）初始化栈指针和帧指针。我们可以看到程序结尾处（第 39 行）声明了标号 Stack，并且用一个.pos 命令来指明了地址 0x100。因此我们的栈会从这个地址开始，向下增长。

程序的第 8～12 行声明了一个四个字的数组，值分别为 0xd、0xc0、0xb00 和 0xa000。标号 array 表明了这个数组的起始，并且在四字节边界处对齐（用.align 命令指定）。第 14～19 行给出了“main”过程，在过程中对那个四个字的数组调用了 Sum 函数，然后停止。

正如从例子看到的那样，用 Y86 写程序要求程序员完成本来通常交给编译器、链接器和运行时系统来完成的任务。幸好我们只用 Y86 来写一些小的程序，对此一些简单的机制就足够了。

图 4.7 是一个我们称为 YAS 的汇编器对图 4.6 中代码进行汇编的结果。为了便于理解，汇编器的输出结果是 ASCII 码格式的。汇编文件中，在有指令或数据的行上，目标代码包含一个地址，后面跟着 1～6 个字节的值。

code/arch/y86-code/asum.yo

```
                         | # Execution begins at address 0
0x000:                   |           .pos 0
0x000: 308600010000      | init:     irmovl Stack, %esp         # Set up Stack pointer
0x006: 308700010000      |           irmovl Stack, %ebp         # Set up base pointer
0x00c: 7024000000        |           jmp Main                   # Execute main program
                         |
                         | # Array of 4 elements
0x014:                   |           .align 4
0x014: 0d000000          | array:    .long 0xd
0x018: c0000000          |           .long 0xc0
0x01c: 000b0000          |           .long 0xb00
0x020: 00a00000          |           .long 0xa000
                         |
0x024: 308004000000      | Main:     irmovl $4,%eax
0x02a: a008              |           pushl %eax       # Push 4
0x02c: 308214000000      |           irmovl array,%edx
0x032: a028              |           pushl %edx       # Push array
0x034: 803a000000        |           call Sum         # Sum(array, 4)
0x039: 10                |           halt
                         |
                         |           # int Sum(int *Start, int Count)
0x03a: a058              | Sum:      pushl %ebp
0x03c: 2045              |           rrmovl %esp,%ebp
```

```
0x03e: 501508000000 |        mrmovl 8(%ebp),%ecx    # ecx = Start
0x044: 50250c000000 |        mrmovl 12(%ebp),%edx   # edx = Count
0x04a: 308000000000 |        irmovl $0, %eax        # sum = 0
0x050: 6222         |        andl %edx,%edx
0x052: 7374000000   |        je End
0x057: 506100000000 | Loop:  mrmovl (%ecx),%esi  # get *Start
0x05d: 6060         |        addl %esi,%eax      # add to sum
0x05f: 308304000000 |        irmovl $4,%ebx      #
0x065: 6031         |        addl %ebx,%ecx      # Start++
0x067: 3083ffffffff |        irmovl $-1,%ebx     #
0x06d: 6032         |        addl %ebx,%edx      # Count--
0x06f: 7457000000   |        jne Loop            # Stop when 0
0x074: b058         | End: popl %ebp
0x076: 90           |        ret
0x100:              |        .pos 0x100
0x100:              | Stack: # The stack goes here
```

code/arch/y86-code/asum.yo

图 4.7 YAS 汇编器的输出

每一行包含一个十六进制的地址，以及字节数在 1~6 之间的目标代码。

我们实现了一个指令集模拟器，称为 YIS。用模拟器运行我们的例子的目标代码，产生下面这样的输出：

```
Stopped in 46 steps at PC = 0x3a. Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x000000f8
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000
Changes to memory:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
```

模拟器只打印出在模拟过程中被改变的寄存器或存储器中的字。左边给出的是原始值（这里它们都是 0），右边的是最后的值。从输出中我们可以看到，寄存器%eax 的值为 0xabcd，即传给子函数 Sum 的四元素数组的和。另外，我们还能看到栈从地址 0x100 开始，向下增长，栈的使用导致了存储器地址 0xf0~0xfc 都发生了变化。

练习题 4.3

根据下面的 C 代码，用 Y86 代码来实现一个递归求和函数 rSum：

```
int rSum(int *Start, int Count)
{
```

```
    if (Count <= 0)
    return 0;
    return *Start + rSum(Start+1, Count-1);
}
```

在一台IA32机器上编译这段C代码，然后再把那些指令翻译成Y86的指令，这样做可能会很有帮助。

练习题 4.4

pushl指令会把栈指针减4，并且将一个寄存器值写入存储器中。当执行pushl %esp指令时，处理器的行为是不确定的，因为要入栈的寄存器会被同一条指令修改。通常有两种约定：①压入%esp的原始值；②压入减了4的%esp的值。

让我们采用和IA32处理器一样的做法来解决这个问题。我们可以通过阅读Intel关于这条指令的文档来了解它们的做法，但更简单的方法是在实际的机器上做个实验。C编译器正常情况下是不会产生这条指令的，所以我们必须用手工生成的汇编代码来完成这一任务。正如3.15节中讲的那样，在一个C程序中插入少量汇编代码的最好方法就是使用GCC的asm特性。下面是我们写的一个测试程序，你会发现与其试图读asm声明，不如读它前面注释中的汇编代码，那样要容易得多。

```
int pushtest()
{
    int rval;
    /* Insert the following assembly code:
       movl %esp,%eax    # Save stack pointer
       pushl %esp        # Push stack pointer
       popl %edx         # Pop it back
       subl %edx,%eax    # 0 or 4
       movl %eax,rval    # Set as return value
    */
   asm("movl %%esp,%%eax;pushl %%esp;popl %%edx;
        subl %%edx,%%eax;movl %%eax,%0"
         : "=r" (rval)
         : /* No Input */
         : "%edx", "%eax");
    return rval;
}
```

在我们的实验中，我们发现函数pushtest返回的是0，这表示在IA32中pushl %esp指令的行为是怎样的呢？

练习题 4.5

对popl %esp指令也有类似的歧义，可以将%esp置为从存储器中读出的值，也可以置为加了4后的栈指针。同对练习题4.4一样，让我们做个实验来确定IA32机器是怎么处理这条指令的，然后我们的Y86机器就采用同样的方法。

```
int poptest(int tval)
{
```

```
    int rval;
    /* Insert the following assembly code:
       pushl tval         # Save tval on stack
       movl %esp,%edx     # Save stack pointer
       popl %esp          # Pop to stack pointer.
       movl %esp,rval     # Set popped value as return value
       movl %edx,%esp     # Restore original stack pointer
    */
    asm("pushl %1; movl %%esp,%%edx; popl %%esp;
         movl %%esp,%0; movl %%edx,%%esp"
         : "=r" (rval)
         : "r" (tval)
         : "%edx");
    return rval;
}
```

我们发现函数总是返回 tval，也就是传进去作为参数的那个值。这表示在 IA32 中 popl %esp 指令的行为是怎样的？还有什么其他 Y86 指令也应该有相同的行为吗？

4.2 逻辑设计和硬件控制语言 HCL

在硬件设计中，电子电路被用来计算位的函数（functions on bits），以及在各种存储器元素中存储位。大多数现代电路技术都是用信号线上的高电压或低电压来表示不同的位值。通常的技术中，逻辑 1 是用 1.0 伏特左右的高电压表示的，而逻辑 0 是用 0.0 伏特左右的低电压表示的。要实现一个数字系统需要三个主要的组成部分：计算位的函数的组合逻辑、存储位的存储器元素，以及控制存储器元素更新的时钟信号。

本节中，我们简要描述这些不同的组成部分。我们还将介绍 HCL（hardware control language，硬件控制语言），我们用这种语言来描述不同处理器设计的控制逻辑。在此我们只是简略地描述 HCL，HCL 完整的参考请见附录 A。

旁注：现代逻辑设计

硬件设计者曾经描绘示意性的逻辑电路图来进行电路设计（最早是用纸和笔画，后来是用计算机图形终端）。现在，大多数设计都是用 HDL 来表达的。HDL 是一种文本表示，看上去和编程语言类似，但是它是用来描述硬件结构而不是程序行为的。最常用的语言是 Verilog，它的语法类似于 C，另一种是 VHDL，它的语法类似于编程语言 Ada。这些语言本来都是用来表示数字电路的模拟模型的。在 20 世纪 80 年代中期，研究者开发出了逻辑合成（logic synthesis）程序，它可以根据 HDL 的描述生成有效的电路设计。现在出现了许多商用的合成程序，它们已经成为产生数字电路的主要技术。从手工设计电路到合成生成的转变就好像从写汇编程序到写高级语言程序，再用编译器来产生机器代码的转变一样。

4.2.1 逻辑门

逻辑门是数字电路的基本计算元素。它们产生的输出，等于它们输入位值的某个布尔函数。图 4.8 给出的是布尔函数 AND、OR 和 NOT 的标准符号，布尔操作的逻辑门下面是对应的 HCL 表达

式。正如你看到的那样，我们采用了 C 中的逻辑运算符的语法（见 2.1.9 节）：AND 用&&表示，OR 用||表示，而 NOT 用!表示。我们不用 C 中的位运算符&、|和~是因为逻辑门只对一个位进行操作，而不是整个字。

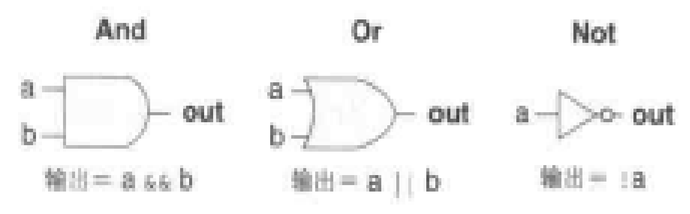


图 4.8 逻辑门类型

每个门产生的输出等于它输入的某个布尔函数。

逻辑门总是活动的（active）。一旦一个门的输入变化了，在很短的时间内，输出就会相应地变化。

4.2.2 组合电路和 HCL 布尔表达式

将很多的逻辑门组合成一个网，我们就能得到计算块（computational block），即组合电路。如何组成这个网有两条限定：

- 两个或多个逻辑门的输出不能接在一起，否则它们可能会使线上的信号矛盾，导致一个不合法的电压或电路故障；
- 这个网必须是无环的。也就是在网中不能有路径经过一系列的门而形成一个回路，这样的回路会导致该网络的计算函数有歧义。

图 4.9 是一个我们觉得非常有用的简单组合电路的例子。它有两个输入 a 和 b，有惟一的输出 eq，当 a 和 b 都是 1（从上面的 AND 门可以看出）或都是 0（从下面的 AND 门可以看出）时，输出为 1。用 HCL 来写这个网的函数就是：

```
bool eq = ( a && b ) || ( !a && !b );
```

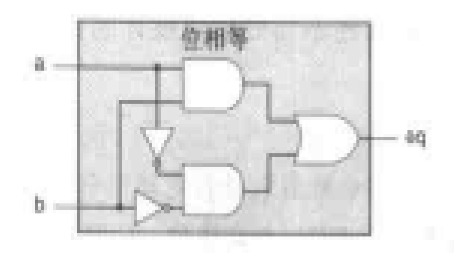


图 4.9 检测位相等的组合电路

当输入都为 0 或都为 1 时，输出等于 1。

这段代码简单地定义了位级（数据类型 bool 表明了这一点）信号 eq，它是输入 a 和 b 的函数。从这个例子可以看出 HCL 使用了 C 风格的语法，"=" 将一个信号名与一个表达式联系起来。不过同 C 不一样，我们不把它看成执行了一次计算并将结果放入存储器中某个位置。相反，它只是用一个名字来称谓一个表达式。

练习题 4.6

写一个信号 xor 的 HCL 表达式，xor 就是异或，输入为 a 和 b。信号 xor 和上面定义的 eq 有什么关系？

图 4.10 给出了另一个简单但很有用的组合电路，称为*多路复用器*（multiplexor）。多路复用器根据输入控制信号的值，从一组不同的数据信号中选出一个。在这个单个位的多路复用器中，两个数据信号是输入位 a 和 b，控制信号是输入位 s。当 s 为 1 时，输出等于 a；而当 s 为 0 时，输出等于 b。在这个电路中，我们可以看出两个 AND 门决定了是否将它们相对应的数据输入传送到 OR 门。当 s 为 0 时，上面的 AND 门将传送信号 b（因为这个门的另一个输入是!s），而当 s 为 1 时，下面的 AND 门将传送信号 a。接下来，我们来写输出信号的 HCL 表达式，使用的就是组合逻辑中相同的操作：

```
bool out = ( s && a ) || ( !s && b );
```

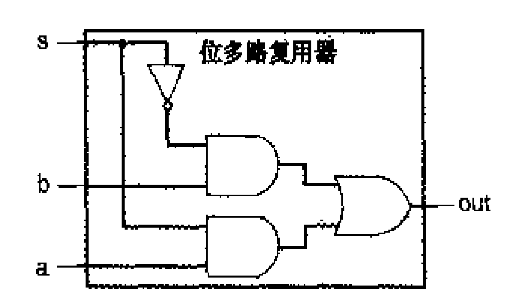


图 4.10 单个位的多路复用器电路

如果控制信号 s 为 1，则输出等于输入 a；当 s 为 0 时，输出等于输入 b。

我们的 HCL 表达式很清楚地表明了组合逻辑电路和 C 中逻辑表达式的相似之处。它们都是用布尔操作来对输入进行计算的函数。值得注意的是，这两种表达计算的方法之间有些区别：

- 因为组合电路是由一些逻辑门组成的，它有个属性就是输出会持续地响应输入的变化。如果电路的输入变化了，在一定的延迟之后，输出也会相应地变化。相比之下，C 表达式只会在程序执行过程中被遇到时才进行求值。
- C 的逻辑表达式允许参数是任意整数，0 表示 FALSE，其他任何值都表示 TRUE。而我们的逻辑门只对位值 0 和 1 进行操作。
- C 的逻辑表达式有个属性就是它们可能只被部分求值。如果一个 AND 或 OR 操作的结果只用对第一个参数求值就能确定，那么就不用对第二个参数求值了。例如，这样一个 C 表达式：

  ```
  ( a && !a ) && func( b, c )
  ```

  这里函数 func 是不会被调用的，因为表达式(a && !a)求值为 0。而组合逻辑没有部分求值这条规则，逻辑门只是简单地响应它们输入的变化。

4.2.3 字级的组合电路和 HCL 整数表达式

通过将逻辑门组成一个更大的网，我们可以构造出能计算更加复杂函数的组合逻辑。通常，我们设计了能对数据字（data words）进行操作的电路。它们是一些位级的信号，代表一个整数或一些控制模式。例如，我们的处理器设计将包括有很多字，字的大小为 4～32 位，代表整数、地址、指令代码和寄存器标识符。

执行字级计算的组合电路是根据输入字的各个位，用逻辑门来计算输出字的各个位。例如图 4.11 中的一个组合电路，它测试两个 32 位字 A 和 B 是否相等。也就是，当且仅当 A 的每一位都和 B 的相应位相等时，输出才为 1。这个电路是用 32 个图 4.9 中所示的那样的单个位相等电路实现的。这

些单个位电路的输出用一个 AND 门连起来，形成了这个电路的输出。

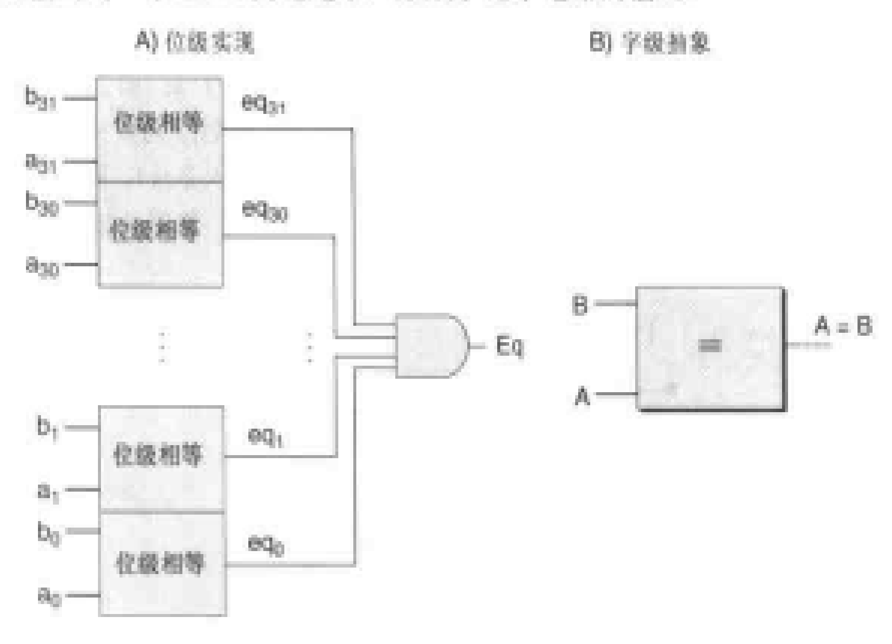


图 4.11 字级相等测试电路

当字 A 的每一位与字 B 中相应的位均相等时，输出等于 1。字级相等是 HCL 中的一个操作。

为了简化，在 HCL 中，我们将所有字级的信号都声明为 int，而不指定字的大小。在特性比较完善的硬件描述语言中，每个字都可以声明有特定的位数。HCL 允许比较字是否相等，因此图 4.11 所示的电路的函数可以在字级上表达成

```
bool Eq = ( A = = B );
```

这里参数 A 和 B 是 int 型的。注意我们使用和 C 中一样的语法习惯，“=” 表示赋值，而 “= =” 是相等运算符。

如图 4.11 中右边所示的那样，在画字级电路的时候，我们用中等粗度的线来表示携带字的单个位的线路，而用虚线来表示布尔信号结果。

练习题 4.7

用练习题 4.6 中的异或电路而不是位级的相等电路来实现一个字级的相等电路。设计一个 32 位字的相等电路需要 32 个字级的异或电路，另外还要两个逻辑门。

图 4.12 给出的是字级的多路复用器电路。这个电路根据控制输入位 s，产生一个 32 位的字 Out，等于两个输入字 A 或者 B 中的一个。这个电路由 32 个相同的子电路组成，每个结构都类似于图 4.10 中的位级多路复用器。不过这个字级的电路并没有简单地复制 32 次位级多路复用器，它只产生一次!s，然后在每个位的地方都重复使用它，从而减少反相器或非门（inverters）的数量。

在我们的处理器中会用到很多种多路复用器。它使得我们能根据某些控制条件，从许多源中选出一个字。在 HCL 中，多路复用函数是用情况（case）表达式来描述的。情况表达式的通用格式如下：

[
  $select_1$: $expr_1$
  $select_2$: $expr_2$

```
        selectk    :    exprk
]
```

这个表达式包含一系列的情况，每种情况 i 都有一个布尔表达式 $select_i$ 和一个整数表达式 $expr_i$，前者表明什么时候该选择这种情况，后者指明的是返回值。

同 C 的开关（switch）语句不同，这里不要求不同的选择表达式之间互斥。从逻辑上讲，这些选择表达式是顺序求值的，且第一个被求值为 1 的情况会被选中。例如，图 4.12 中的字级多路复用器用 HCL 来描述就是：

```
int Out = [
        s: A;
        1: B;
];
```

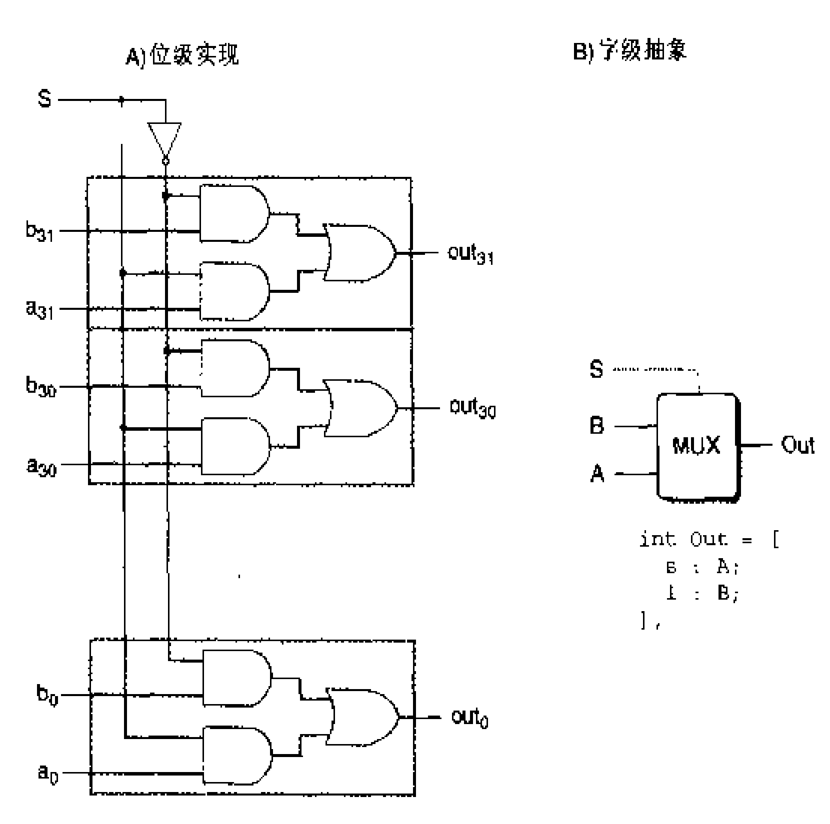


图 4.12 字级多路复用器电路

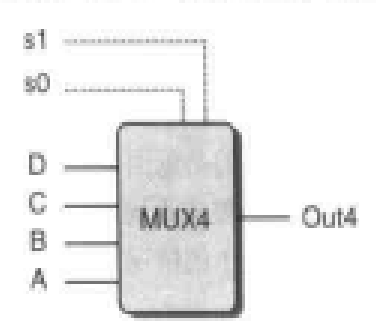
当控制信号 s 为 1 时，输出会等于输入字 A，否则等于 B。HCL 中是用情况（case）表达式来描述多路复用器的。

在这段代码中，第二个选择表达式就是 1，表明如果前面没有情况被选中，那就选择这种情况。这是 HCL 中一种指定默认情况的方法。几乎所有的情况表达式都是以此结尾的。

允许不互斥的选择表达式使得 HCL 代码的可读性更好。实际的硬件多路复用器的信号必须互斥，它们要控制哪个输入字应该被传送到输出，就像图 4.12 中的信号 s 和!s。要将一个 HCL 情况表达式翻译成硬件，逻辑合成（logic synthesis）程序需要分析选择表达式集合，并解决任何可能的冲突，确保只有第一个满足的情况才会被选中。

选择表达式可以是任意的布尔表达式，且有任意多的情况（case）。这就使得情况表达式能描述

带复杂选择标准的、多种输入信号的块。例如，考虑下面这个四路复用器的图：

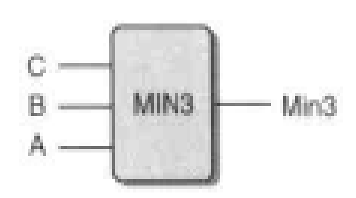


这个电路根据控制信号 s1 和 s0，从四个输入字 A、B、C 和 D 中选择一个，这里用一个两位的二进制数作为控制信号。我们可以用 HCL 来表示这个电路，用布尔表达式描述控制位模式的不同组合：

```
int Out4 = [
        !s1 && !s0  : A;    # 00
        !s1         : B;    # 01
        !s0         : C;    # 10
        1           : D;    # 11
];
```

右边的注释（任何以#开头到行尾结束的文字都是注释）表明了 s1 和 s0 的什么组合会导致该种情况会被选中。可以看到选择表达式有时可以简化，因为只有第一个匹配的情况才会被选中。例如，第二个表达式可以写成!s1，而不用写得更完整!s1 && s0，因为另一种可能 s1 等于 0 已经出现在了第一个选择表达式中了。

让我们来看最后一个例子，假设我们想设计一个逻辑电路来找一组字 A、B 和 C 中的最小值，如下图所示：



用 HCL 来表达就是：

```
int Min3 = [
        A <= B && A <= C  : A;
        B <= A && B <= C  : B;
        1                 : C;
];
```

练习题 4.8

写这样一个电路的 HCL 代码，对于输入字 A、B 和 C，它选择中间值。也就是，输出等于三个输入中居于最小值和最大值之间的那个字。

组合逻辑电路可以设计成在字级数据上执行许多不同类型的操作。具体的设计已经超出了我们讨论的范围。算术/逻辑单元（ALU）是一种很重要的组合电路，图 4.13 是它的一个抽象的图示。

这个电路有三个输入：两个标号为 A 和 B 的数据输入，以及一个控制输入。根据控制输入的设置，电路会对数据输入执行不同的算术或逻辑操作。这个 ALU 中画的四个操作对应于 Y86 指令集支持的四种不同的整数操作，而控制值和这些操作的功能码相对应（图 4.3）。我们还注意到减法的操作数顺序，是输入 A 减去输入 B。之所以这样做，是为了使这个顺序与 subl 指令的参数顺序一致。

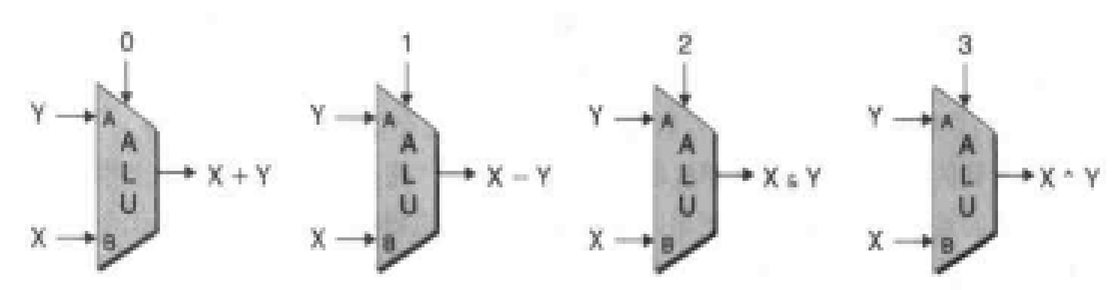
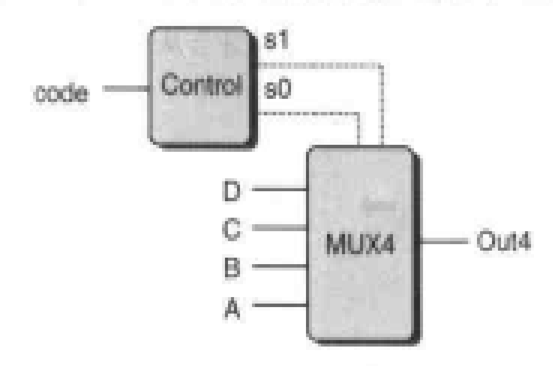


图 4.13 算术/逻辑单元（ALU）

根据函数输入的设置，该电路会执行四种算术和逻辑运算中的一种。

4.2.4 集合关系（Set Membership）

在我们的处理器设计中，很多时候都需要将一个信号与许多可能匹配的信号做比较，以此来检测正在处理的某些指令代码是否属于某一类指令代码。下面来看一个简单的例子，我们想从一个两位信号代码中选择高位和低位来为图 4.12 中的四路复用器产生信号 s1 和 s0，如下图所示：



在这个电路中，两位的信号代码可以用来控制对四个数据字 A、B、C 和 D 的选择。根据可能的代码值，可以用相等测试来表示信号 s1 和 s0 的产生：

```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```

还有一种更简洁的方式来表示当 code 在集合{2, 3}中时 s1 为 1，而 code 在集合{1, 3}中时 s0 为 1：

```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```

判断集合关系的通用格式是：

$$iexpr \text{ in } \{iexpr_1, iexpr_2, \cdots, iexpr_k\}$$

这里被测试的值 *iexpr* 和待匹配的值 $iexpr_1$～$iexpr_k$ 都是整数表达式。

4.2.5 存储器和时钟控制

组合电路从本质上讲，不存储任何信息。相反，它们只是简单地响应输入信号，产生等于输入

的某个函数输出。为了产生时序电路（sequential circuit），也就是有状态并且在这个状态上进行计算的系统，我们必须引入按位存储信息的设备。我们考虑两类存储器设备：

- 时钟寄存器（简称寄存器）存储单个位或字。时钟信号控制寄存器加载输入值。
- 随机访问存储器（简称存储器）存储多个字，用地址来选择该读或该写哪个字。随机访问存储器的例子包括：处理器的虚拟存储器系统，硬件和操作系统软件结合起来使处理器可以在一个很大的地址空间内访问任意的字；寄存器文件，在此，寄存器标识符作为地址。在IA32或Y86处理器中，寄存器文件有八个程序寄存器（%eax、%ecx等）。

正如我们看到的那样，在讲硬件和机器级编程时，单词"寄存器"有些细微的差别。在硬件中，寄存器直接将它的输入和输出线连接到电路的其他部分。在机器级编程中，寄存器代表的是CPU中为数不多的可寻址的字，这里的地址是寄存器ID。这些字通常都存在寄存器文件中，虽然我们会看到硬件有时可以直接将一个字从一个指令传送到另一个指令，以避免先写寄存器文件再读出来的延迟。需要避免歧义时，我们会分别称呼这两类寄存器为"硬件寄存器"和"程序寄存器"。

图4.14给出了一个硬件寄存器，以及它是如何工作的。大多数时候，寄存器都保持在稳定状态（用x表示），产生的输出等于它的当前状态。信号沿着寄存器前面的组合逻辑传播，这时，产生了一个新的寄存器输入（用y表示），但只要时钟是低电位，寄存器的输出就仍保持不变。当时钟变成高电位的时候，输入信号就加载到寄存器，成为下一个状态 y，这个状态就成为寄存器的新输出，直到下一个时钟上升沿（rising clock edge）的时候。特别指出的是寄存器被作为电路不同部分中的组合逻辑之间的屏障。只有在每个时钟上升沿时，值才会从寄存器的输入传送到输出。

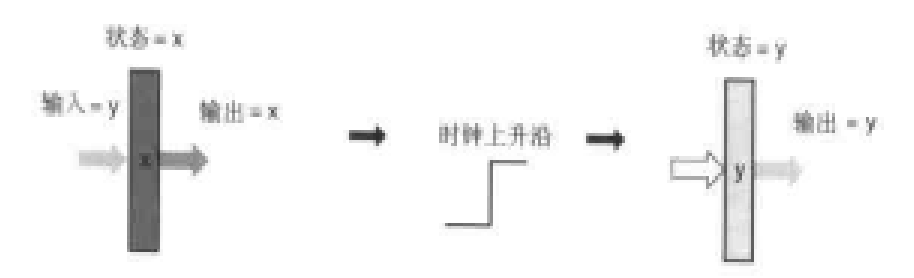
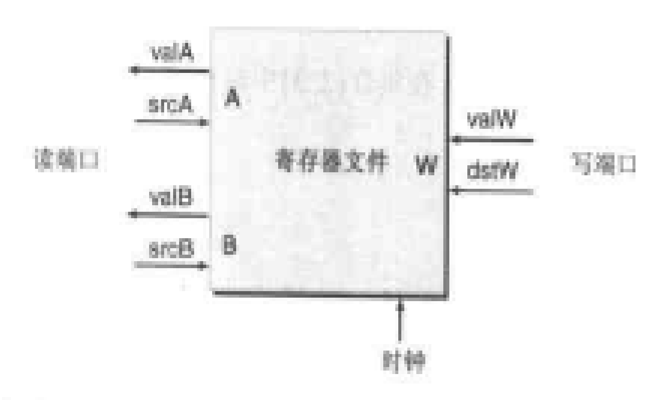


图4.14 寄存器操作

寄存器输出会一直保持在当前寄存器状态上，直到时钟信号上升。当时钟上升时，寄存器输入上的值会成为新的寄存器状态。

下面的图展示了一个典型的寄存器文件：



寄存器文件有两个读端口（A和B），还有一个写端口（W）。这样一个多端口随机访问存储器允许同时进行多个读和写操作。在图中所示的寄存器文件中，电路可以读两个程序寄存器的值，同

时更新第三个寄存器的状态。每个端口都有一个地址输入，表明该选择哪个程序寄存器，另外还有一个数据输出或对应该程序寄存器的输入值。地址是用图 4.4 中编码表示的寄存器标识符。两个读端口有地址输入 srcA 和 srcB（“source A”和“source B”的缩写）和数据输出 valA 和 valB（“value A”和“value B”的缩写）。写端口有地址输入 dstW（“destination W”的缩写），以及数据输入 valW（“value W”的缩写）。

虽然寄存器文件不是组合电路(因为它有内部的存储)，但是从中读取字的操作与以地址为输入、数据为输出的一块组合逻辑是一样的。当 srcA 或 srcB 被设成某个寄存器 ID 时，在一段延迟之后，相应程序寄存器的值就会出现在 valA 或 valB 上。例如，将 srcA 设为 3，就会读程序寄存器%ebx 的值，然后这个值就会出现在输出 valA 上。

时钟信号按照类似于将值加载进时钟寄存器一样的方式控制向寄存器文件写入字。每次时钟上升时，输入 valW 上的值被写入输入 dstW 上的寄存器 ID 指示的程序寄存器。当 dstW 设为特殊的 ID 值 8 时，不会写任何程序寄存器。

4.3 Y86 的顺序（sequential）实现

现在我们已经有了实现 Y86 处理器所需要的部件。首先，我们讲一个称为 SEQ(取的是“sequential”处理器的意思）的处理器。每个时钟周期上，SEQ 执行用来处理一条完整指令所需的所有步骤。不过这需要一个很长的时钟周期时间，因此时钟周期频率会低到不可接受。我们开发 SEQ 的目标就是提供实现我们最终目的的第一步，我们的最终目的是实现一个高效的、流水线化的处理器。

4.3.1 将处理组织成阶段

通常，处理一条指令包括很多操作。我们将它们组织成某个特殊的阶段序列，使得即使指令的动作差异很大，但所有的指令都遵循统一的序列。每一步的具体处理取决于正在执行的指令。创建这么一个框架使我们能够设计一个能充分利用硬件的处理器。下面是关于各个阶段以及各阶段内执行操作的简略描述：

- 取指（fetch）：取指阶段从存储器读入指令，地址为程序计数器（PC）的值。从指令中抽取出指令指示符字节的两个四位部分，称为 icode（指令代码）和 ifun（指令功能）。它可能取出一个寄存器指示符字节，指明一个或两个寄存器操作数指示符 rA 和 rB。它还可能取出一个四字节常数字 valC。它按顺序方式计算当前指令的下一条指令的地址 valP。也就是说，valP 等于 PC 的值加上已取出指令的长度。
- 解码（decode）：解码阶段从寄存器文件读入最多两个操作数，得到值 valA 和/或 valB。通常，它读入指令 rA 和 rB 字段指明的寄存器，不过有些指令是读寄存器%esp 的。
- 执行（execute）：在执行阶段，算术/逻辑单元（ALU）要么执行指令指明的操作（根据 ifun 的值），计算存储器引用的有效地址，要么增加或减少栈指针。我们称得到的值为 valE。在此，也可能设置条件码。对一条跳转指令来说，这个阶段会检验条件码和（ifun 给出的）分支条件，看是不是应该选择分支。
- 访存(memory)：访存阶段可以将数据写入存储器，或者从存储器读出数据。读出的值为 valM。
- 写回（write back）：写回阶段最多可以写两个结果到寄存器文件。

- 更新PC（PC update）：将PC设置成下一条指令的地址。

处理器无限制地循环执行这些阶段，只有在遇到halt指令或一些错误情况时，才会停下来。我们处理的错误情况包括非法存储器地址（程序地址或数据地址），以及非法指令。

从前面的讲述可以看出，执行一条指令是需要进行很多处理的。不仅要执行指令所表明的操作，还要计算地址、更新栈指针，以及确定下一条指令的地址。幸好每条指令的整个流程都比较相似。因为我们想使硬件数量尽可能的少，并且最终将把它映射到一个二维的集成电路芯片的表面，一个非常简单而一致的结构是非常重要的。降低复杂度的一种方法是让不同的指令共享尽量多的硬件。例如，我们的每个处理器设计都只含有一个算术/逻辑单元，根据所执行的指令类型的不同，它的使用方式也不同。在硬件上复制逻辑块的成本比软件中有重复代码的成本大得多，而且在硬件系统中处理许多特殊情况和特性要比用软件来处理困难得多。

我们面临的一个挑战是将每条不同指令所需要的计算放入到上述那个通用框架中。我们会使用图4.15中所示的代码来描述不同Y86指令的处理。图4.16～图4.19中的表描述了不同Y86指令在各个阶段是怎样处理的。要好好研究一下这些表，表中的这种格式很容易映射到硬件。这些表中的每一行都描述了一个信号或存储状态的分配（用分配操作←来表示）。阅读时可以把它看成是从上至下的顺序求值。后面我们将这些计算映射到硬件时，会发现其实并不需要严格按照顺序来执行这些求值。

```
1   0x000: 308209000000 |     irmovl $9, %edx
2   0x006: 308315000000 |     irmovl $21, %ebx
3   0x00c: 6123         |     subl %edx, %ebx         # subtract
4   0x00e: 308480000000 |     irmovl $128,%esp        # Practice Prob. 4.9
5   0x014: 404364000000 |     rmmovl %esp, 100(%ebx)  # store
6   0x01a: a028         |     pushl %edx              # push
7   0x01c: b008         |     popl %eax               # Practice Prob. 4.10
8   0x01e: 7328000000   |     je done                 # Not taken
9   0x023: 8029000000   |     call proc               # Practice Prob. 4.13
10  0x028:              | done:
11  0x028: 10           |     halt
12  0x029:              | proc:
13  0x029: 90           |     ret                     # Return
```

图4.15 Y86指令序列示例

我们会通过各个阶段来跟踪这些指令的处理。

图4.16给出了OPl（整数和逻辑运算）、rrmovl（寄存器-寄存器传送）和irmovl（立即数-寄存器传送）类型的指令所需的处理。让我们先来考虑一下整数操作。回顾图4.2，可以看到我们小心地选择了指令编码，这样四个整数操作（addl、subl、andl和xorl）有着相同的icode值。我们可以以相同的步骤顺序来处理它们，除了ALU计算必须根据ifun中编码的具体的指令操作来设定。

整数操作指令的处理遵循上面列出的通用模式。在取指阶段，我们不需要常数字，所以valP的计算就是PC + 2。在解码阶段，我们要读两个操作数。在执行阶段，它们和功能指示符ifun一起再提供给ALU，然后valE内放入指令结果。这个计算是用表达式valB OP valA来表达的，这里OP代表ifun指定的操作。要注意两个参数的顺序——这个顺序与Y86（和IA32）的习惯是一致的。例如，指令subl %eax, %edx，计算的是R[%edx] - R[%eax]的值。这些指令在访存阶段什么也不做，而在写回阶段，valE被写入寄存器rB，然后PC设为valP，整个指令的执行就结束了。

阶段	opl rA, rB	rrmovl rA, rB	irmovl V, rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6
解码	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
执行	valE ← valB OP valA Set CC	valE ← 0+valA	valE ← 0+valC
访存			
写回	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE
更新 PC	PC ← valP	PC ← valP	PC ← valP

图 4.16 Y86 指令 OPl、rrmovl 和 irmovl 在顺序实现中的计算

这些指令计算了一个值，并将结果存放在寄存器中。符号 icode:ifun 表明指令字节的两个组成部分，而 rA:rB 表明寄存器指示符字节的两个组成部分。符号 $M_1[x]$表示访问（读或者写）存储器位置 x 处的一个字节，而 $M_4[x]$表示访问四个字节。

旁注：跟踪 subl 指令的执行

作为一个例子，让我们来看看一条 subl 指令的处理过程，这条指令是图 4.15 所示目标代码的第 3 行中的 subl 指令。我们可以看到前面两条指令分别将寄存器%edx 和%ebx 初始化成 9 和 21。还能看到指令是位于地址 0x00c，有两个字节，值分别为 0x61 和 0x23。这条指令的处理如下图所示，左边列出了处理一个 OPl 指令的通用的规则（图 4.16），而右边列出的是对这条指令的计算。

阶段	通用	具体
	OPl rA,rB	subl %edx, %edx
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[0x00c]=6:1 rA:rB ← M_1[0x00d]=2:3 valP ← 0x00c+2=0x00e
解码	valA ← R[rA] valB ← R[rB]	valA ← R[%edx]=9 valB ← R[%ebx]=21
执行	valE ← valB OP valA Set CC	valE ← 21-9=12 ZF ← 0, SF ← 0, OF ← 0
访存		
写回	R[rB] ← valE	R[%ebx] ← valE=12
更新 PC	PC ← valP	PC ← valP=0x00e

就像这个记录表明的那样，我们达到了理想的目标，寄存器%ebx 设成了 12，三个条件码都设成了 0，而 PC 加了 2。

执行 rrmovl 指令和执行算术运算类似。不过，不需要取第二个寄存器操作数。我们将 ALU 的

第二个输入设为 0，先把它和第一个操作数相加，得到 valE = valA，然后再把这个值写到寄存器文件。对 irmovl 的处理与此类似，除了 ALU 的第一个输入为常数值 valC。另外，因为是长指令格式，对于 irmovl，程序计数器必须加 6。所有这些指令都不改变条件码。

练习题 4.9

填写下表的右边一栏，这个表描述的是图 4.15 中目标代码第 4 行上的 irmovl 指令的处理情况：

阶段	通用	具体
	`irmovl` V, rB	`irmovl $128, %esp`
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	
解码		
执行	valE ← 0+valC	
访存		
写回	R[rB] ← valE	
更新 PC	PC ← valP	

这条指令的执行会怎样改变寄存器和 PC 呢？

图 4.17 给出了存储器读写指令 rmmovl 和 mrmovl 所需要的处理。基本流程也和前面一样，不过是用 ALU 来加 valC 和 valB，得到存储器操作的有效地址（位移量与基址寄存器值之和）。在访存阶段，会将寄存器值 valA 写到存储器，或者从存储器中读出 valM。

阶段	`rmmovl` rA, D(rB)	`mrmovl` D(rB), rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6
解码	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
执行	valE ← valB+valC	valE ← valB+valC
访存	M_4[valE] ← valA	valM ← M_4[valE]
写回		R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

图 4.17 Y86 指令 rmmovl 和 mrmovl 在顺序实现中的计算

这些指令读或者写存储器。

旁注：跟踪 rmmovl 指令的执行

让我们来看看图 4.15 中目标代码的第 5 行上 rmmovl 指令的处理情况。可以看到，前面的指令已将寄存器%esp 初始化成了 128，而%ebx 仍然是 subl 指令（第三行）算出来的结果 12。我们可以看到，指令位于地址 0x014，有六个字节，前两个的值为 0x40 和 0x43，后四个是数字 0x00000064（十进制数 100）按字节反过来得到的数。各个阶段的处理如下：

阶段	通用	具体
	rmmovl rA, D(rB)	rmmovl %esp, 100(%ebx)
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:ifun ← M_1[0x014]=4:0 rA:rB ← M_1[0x015]=4:3 valC ← M_4[0x016]=100 valP ← 0x014+6=0x01a
解码	valA ← R[rA] valB ← R[rB]	valA ← R[%esp]=128 valB ← R[%ebx]=12
执行	valE ← valB + valC	valE ← 12+100=112
访存	M_4[valE] ← valA	M_4[112] ← 128
写回		
更新 PC	PC ← valP	PC ← 0x01a

就像这个记录表明的那样，这条指令的效果就是将 128 写入存储器地址 112，并将 PC 加 6。

图 4.18 给出了处理 pushl 和 popl 指令所需的步骤。它们可以算是最难实现的 Y86 指令了，因为它们既涉及到访问存储器，又要增加或减少栈指针。虽然这两条指令的流程比较相似，但是它们还是有很重要的区别的。

阶段	pushl rA	popl rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
解码	valA ← R[rA] valB ← R[%esp]	valA ← R[%esp] valB ← R[%esp]
执行	valE ← valB+(-4)	valE ← valB+4
访存	M_4[valE] ← valA	valM ← M_4[valA]
写回	R[%esp] ← valE	R[%esp] ← valE R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

图 4.18 Y86 指令 pushl 和 popl 在顺序实现中的计算

这些指令将值压入或弹出栈。

pushl 指令开始时很像我们前面讲过的指令，但是在解码阶段，是用%esp 作为第二个寄存器操作数的标识符，将栈指针赋值为 valB。在执行阶段，用 ALU 将栈指针减 4。减过 4 的值就是存储器写的地址，在写回阶段还会存回到%esp 中。将 valE 作为写操作的地址，是遵循了 Y86（和 IA32）的惯例，也就是在写之前，pushl 应该先将栈指针减去 4，即使栈指针的更新实际上是在存储器操作完成之后才进行的。

旁注：跟踪 pushl 指令的执行

让我们来看看图 4.15 中目标代码的第 6 行上 rmmovl 指令的处理情况。此时，寄存器%edx 的值为 9，而寄存器%esp 的值为 128。我们还可以看到指令是位于地址 0x01a，有两个字节，值分别为 0xa0 和 0x28。各个阶段的处理如下：

阶段	通用	具体
	pushl rA	pushl %edx
取指	icode:ifun ← M₁[PC] rA:rB ← M₁[PC+1] valP ← PC+2	icode:ifun ← M₁[0x00c]=a:0 rA:rB ← M₁[0x01d]=2:8 valP ← 0x01a+2=0x01c
解码	valA ← R[rA] valB ← R[%esp]	valA ← R[%edx]=9 valB ← R[%esp]=128
执行	valE ← valB+(-4)	valE ← 128+(-4)=124
访存	M₄[valE] ← valA	M₄[124] ← 9
写回	R[%esp] ← valE	R[%esp] ← 124
更新 PC	PC ← valP	PC ← 0x01c

就像这个记录表明的那样，这条指令的效果就是将%esp 设为 124，将 9 写入地址 124，并将 PC 加 2。

popl 指令的执行与 pushl 的执行类似，除了在解码阶段要读两次栈指针以外。这样做看上去是很多余，但是我们会看到让 valA 和 valB 都存放栈指针的值，会使后面的流程跟其他的指令更相似，增强了设计的整体一致性。在执行阶段，用 ALU 给栈指针加 4，但是用没加过 4 的原始值作为存储器操作的地址。在写回阶段，要用加过 4 的栈指针更新栈指针寄存器，还要将寄存器 rA 更新为从存储器中读出的值。用没加过 4 的值作为存储器读地址，保持了 Y86（和 IA32）的惯例，popl 应该首先读存储器，然后再增加栈指针。

练习题 4.10

填写下表的右边一栏，这个表描述的是图 4.15 中目标代码第 7 行上的 popl 指令的处理情况：

阶段	通用	具体
	popl rA	popl %eax
取指	icode:ifun ← M₁[PC] rA:rB ← M₁[PC+1] valP ← PC+2	
解码	valA ← R[%esp] valB ← R[%esp]	
执行	valE ← ValB+4	
访存	valM ← M₄[valA]	
写回	R[%esp] ← valE R[rB] ← valM	
更新 PC	PC ← valP	

这条指令的执行会怎样改变寄存器和 PC 呢？

练习题 4.11

根据图 4.18 中列出的步骤，指令 pushl %esp 会有什么样的效果？这与练习题 4.4 中确定的 Y86 期望的行为一致吗？

练习题 4.12

假设 popl 在写回阶段中的两个寄存器写操作按照图 4.18 列出的顺序进行，popl %esp 执行的效果会是怎样的？这与练习题 4.5 中确定的 Y86 期望的行为一致吗？

图 4.19 表明了我们的三类控制转移指令的处理：各种跳转、call 和 ret。可以看到，我们能用同前面指令一样的整体流程来实现这些指令。

阶段	jXX Dest	call Dest	ret
取指	icode:ifun ← M₁[PC] valC ← M₄[PC+1] valP ← PC+5	icode:ifun ← M₁[PC] valC ← M₄[PC+1] valP ← PC+5	icode:ifun ← M₁[PC] valP ← PC+1
解码		valB ← R[%esp]	valA ← R[%esp] valB ← R[%esp]
执行	Bch ← Cond(CC,ifun)	valE ← valB+(-4)	valE←valB+4
访存		M₄[valE] ← valP	valM ← M₄[valA]
写回		R[%esp] ← valE	R[%esp] ← valE
更新 PC	PC ← Bch? valC:valP	PC ← valC	PC ← valM

图 4.19 Y86 指令 jxx、call 和 ret 在顺序实现中的计算

这些指令导致控制转移。

同对整数操作一样，我们能够以一种统一的方式处理所有的跳转指令，因为它们的不同只在于判断是否要选择分支的时候。跳转指令在取指和解码阶段都和前面讲的其他指令类似，除了它不需要寄存器指示符字节以外。在执行阶段，我们检查条件码和跳转条件来确定是否要选择分支，产生出一个一位信号 Bch。在更新 PC 阶段，我们检查这个标志，如果这个标志为 1，就将 PC 设为 valC（跳转目标），如果为 0，就设为 valP（下一条指令的地址）。我们的表示 *x?a:b* 类似于 C 中的条件表达式——当 *x* 非零时，它等于 *a*，当 *x* 为零时，等于 *b*。

旁注：跟踪 je 指令的执行

让我们来看看图 4.15 中目标代码的第 8 行上 je 指令的处理情况。subl 指令（第 3 行）已经将所有的条件码都置为了 0，所以不会选择分支。该指令位于地址 0x01e，有 5 个字节。第一个字节的值为 0x73，而剩下的四个字节是数字 0x00000028 按字节反过来得到的数，也就是跳转的目标。各个阶段的处理如下：

阶段	通用	具体
	jxx Dest	je 0x028
取指	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	icode:ifun ← M_1[0x01e]=7:3 valC ← M_4[0x01f]=0x028 valP ← 0x01e+5=0x023
解码		
执行	Bch ← Cond(CC, ifun)	Bch ← Cond(<0,0,0>,3)=0
访存		
写回		
更新 PC	PC ← Bch? valC:valP	PC ← 0?0x028:0x023 = 0x023

就像这个记录表明的那样，这条指令的效果就是将 PC 加 5。

指令 call 和 ret 与指令 pushl 和 popl 类似，除了要将程序计数器的值入栈和出栈以外。对指令 call，我们要将 valP，也就是 call 指令后紧跟着的那条指令的地址，压入栈中。在更新 PC 阶段，将 PC 设为 valC，也就是调用的目的地。对指令 ret，在更新 PC 阶段，我们将 valM，从栈中取出的值，赋值给 PC。

练习题 4.13

填写下表的右边一栏，这个表描述的是图 4.15 中目标代码第 9 行上的 call 指令的处理情况：

阶段	通用	具体
	call Dest	call 0x029
取指	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	
解码	 valB ← R[%esp]	
执行	valE ← ValB+(-4)	
访存	M_4[valE] ← valP	
写回	R[%esp] ← valE	
更新 PC	PC ← valc	

这条指令的执行会怎样改变寄存器、PC 和存储器呢？

旁注：跟踪 ret 指令的执行

让我们来看看图 4.15 中目标代码的第 13 行上 ret 指令的处理情况。指令的地址是 0x029，只有一个字节的编码，0x90。前面的 call 指令将%esp 置为了 124，并将返回地址 0x028 存放在了存储器地址 124。各个阶段的处理如下：

阶段	通用	具体
	ret	ret
取指	icode:ifun ← M_1[PC] valP ← PC+1	icode:ifun ← M_1[0x029]=9:0 valP ← 0x029+5=0x02a
解码	valA ← R[%esp] valB ← R[%esp]	valA ← R[%esp]=124 valB ← R[%esp]=124
执行	valE ← valB+4	valE ← 124+4=128
访存	valM ← M_4[valA]	valM ← M_4[124]=0x028
写回	R[%esp] ← valE	R[%esp] ← 128
更新 PC	PC ← valM	PC ← 0x028

就像这个记录表明的那样，这条指令的效果就是将PC设为0x028，halt指令的地址。同时也将%esp置为了128。

我们创建了一个统一的框架，能处理所有不同类型的Y86指令。虽然指令的行为各不相同，但是我们可以将指令的处理组织成六个阶段。现在我们的任务是创建硬件设计来实现这些阶段，并把它们连接起来。

4.3.2 SEQ硬件结构

实现所有Y86指令所需要的计算可以被组织成六个基本阶段：取指、解码、执行、访存、写回和更新PC。图4.20给出了一个能执行这些计算的硬件结构的抽象表示。程序计数器放在寄存器中，在图中左下角（标了"PC"）。信息沿着线流动（多条线重合就用宽一点的灰线来表示），先向上，再向右。同各个阶段相关的*硬件单元*（hardware units）负责执行这些处理。反馈线路向下回到右边，包括要写到寄存器文件的更新值，以及更新的程序计数器值。这张图省略了一些小的组合逻辑块，还省略了所有用来操作各个硬件单元以及将相应的值路由到这些单元的控制逻辑。待会儿我们会详细讲述这个问题。我们从下往上画处理器和流程的方法似乎有点奇怪。在我们开始设计流水线化的处理器时，我们会解释这么画的原因。

硬件单元与各个处理阶段相关联：

取指：将程序计数器寄存器作为地址，指令存储器读取一个指令的字节。PC增加器（PC incrementer）计算valP，即增加了的程序计数器。

解码：寄存器文件有两个读端口A和B，从这两个端口同时读寄存器值valA和valB。

执行:执行阶段会根据指令的类型，将算术/逻辑单元（ALU）用于不同的目的。对整数操作，它要执行指令所指定的运算。对其他指令，它会作为一个加法器来计算增加或减少栈指针，或者计算有效地址，或者只是简单地加0，将一个输入传递到输出。

条件码寄存器（CC）有三个条件码位。ALU负责计算条件码的新值。当执行一条跳转指令时，会根据条件码和跳转类型来计算分支信号Bch。

访存：在执行访存操作时，数据存储器（data memory）读出或写入一个存储器字。指令和数据存储器访问的是相同的存储器位置，但是用于不同的目的。

写回：寄存器文件有两个写端口。端口E用来写ALU计算出来的值，而端口M用来写从数据存储器中读出的值。

图4.21更详细地给出了实现SEQ所需要的硬件（虽然到分析每个阶段时，我们才会看到完整的细节）。我们看到一组和前面一样的硬件单元，但是现在线路看得更清楚了。在这幅图以及我们其他的硬件图中，都使用的是下面的作图惯例。

- *用带淡点的浅灰色方框表示硬件单元*。这包括存储器、ALU等等。在我们所有的处理器实现中，都会使用这一组基本的单元。我们把这些单元看成"黑盒子"，不关心它们的细节设计。
- *控制逻辑块是用灰色圆角矩形表示的*。这些块用来从一组信号源中进行选择，或者用来计算一些布尔函数。我们会详细分析这些块的，包括详细说明HCL描述。
- *线路的名字在白色圆角方框中说明*。它们只是线路的标识，而不是什么硬件元素。
- *宽度为字长的数据连接用中等粗度的线表示*。每条这样的线实际上都代表一簇32根线，

并列地连在一起，将字从硬件的一个部分传送到另一部分。

图 4.20 SEQ 的抽象视图，一种顺序实现

指令执行过程中的信息处理沿着顺时针方向进行，从用程序计数器（PC）取指令开始，如图中左下角所示。

- 宽度为字节或更窄的数据连接用细线表示。根据线上要携带的值的类型，每条这样的线实际上都代表一簇 4 根或 8 根线。
- 单个位的连接用点线来表示。这代表芯片上单元与块之间传递的控制值。

图 4.16～图 4.19 中所有的计算都有这样的性质，每一行都代表某个值的计算，如 valP，或者激活某个硬件单元，如存储器。图 4.22 的第二栏列出了这些计算和动作。除了我们已经讲过的那些信

号以外，还列出了四个寄存器 ID 信号：srcA，valA 的源；srcB，valB 的源；dstE，写入 valE 的寄存器；以及 dstM，写入 valM 的寄存器。

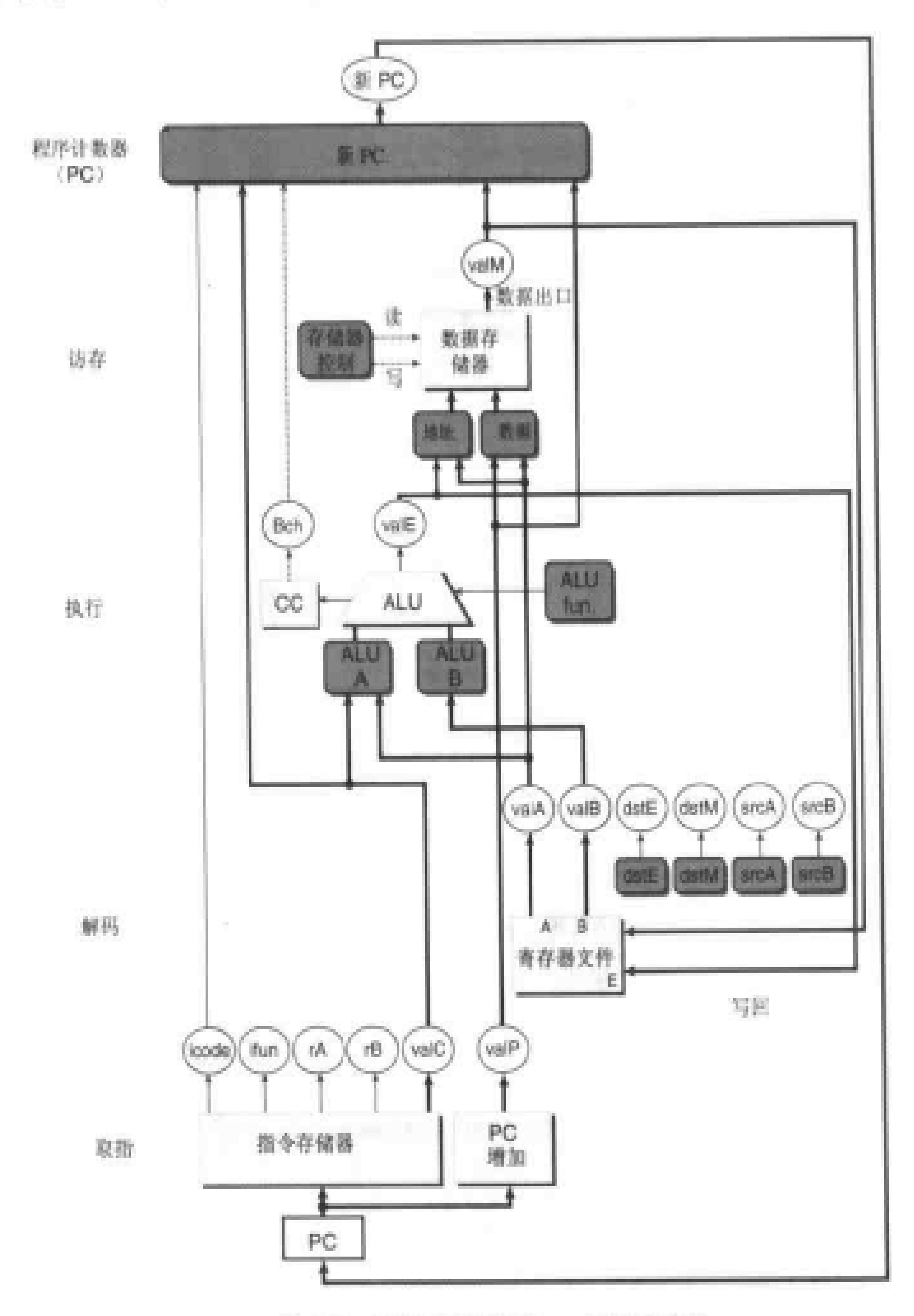


图 4.21 SEQ 的硬件结构，一种顺序实现

有的控制信号以及寄存器和控制字连接，都没有画出来。

阶段	计算	OP1 rA, rB	mrmovl D(rB), rA
取指	icode, ifun rA, rB valC valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6
解码	valA, srcA valB, srcB	valA ← R[rA] valB ← R[rB]	 valB ← R[rB]
执行	valE Cond.codes	valE ← valB OP valA Set CC	valE ← valB+valC
访存	read/write		valM ← M_4[valE]
写回	E port, dstE M port, dstM	R[rB] ← valE	 R[rA] ← valM
更新PC	PC	PC ← valP	PC ← valP

图 4.22 标识顺序实现中的不同计算步骤

第二栏标识出 SEQ 的阶段中正在被计算的值，或正在被执行的操作。作为示例给出的是指令 OPl 和 mrmovl 的计算。

图中，右边两栏给出的是指令 OPl 和 mrmovl 的计算，来说明要计算的值。要将这些计算映射到硬件上，我们要实现控制逻辑，它能在不同硬件单元之间传送数据，以及操作这些单元（即对每个不同的指令执行指定的运算）。这就是控制逻辑块的目标，控制逻辑块在图 4.21 中用灰色圆角方框表示。我们的任务就是着手每个阶段，创建出这些块的详细设计。

4.3.3 SEQ 的时序（timing）

在介绍图 4.16～图 4.19 时，我们说过阅读的时候要把它们看成是用程序符号写的，那些赋值是从上到下顺序执行的。然而，图 4.21 中硬件结构的操作运行根本完全不同。让我们来看看这些硬件是怎样实现表中列出的那些行为的。

我们的 SEQ 的实现包括组合逻辑和两种存储器设备：时钟控制的寄存器（程序计数器和条件码寄存器）和随机访问存储器（寄存器文件、指令存储器和数据存储器）。组合逻辑不需要任何定序（sequencing）或控制——只要输入变化了，值就通过逻辑门网络传播。正如我们提到过的那样，我们将读随机访问存储器看成和组合逻辑一样的操作，根据地址输入产生输出字。因为我们的指令存储器只用来读指令，因此我们可以将这个单元看成组合逻辑。

现在还剩四个硬件单元需要对它们的定序（sequencing）进行明确的控制——程序计数器、条件码寄存器、数据存储器和寄存器文件。这些单元是通过一个时钟信号来控制的，它触发将新值装载到寄存器以及将值写到随机访问存储器。每个时钟周期，程序计数器都会装载新的指令地址。只有在执行整数运算指令时，才会装载条件码寄存器。只有在执行 rmmovl、pushl 或 call 指令时，才会写数据存储器。寄存器文件的两个写端口允许每个时钟周期更新两个程序寄存器，不过我们可以用特殊的寄存器 ID 8 作为端口地址，来表明在此端口不应该执行写操作。

控制我们处理器中活动的定序（sequencing），只需要寄存器和存储器的时钟控制。我们的硬件获得了就好像图 4.16～图 4.19 中那些赋值顺序执行一样的效果，即使所有的状态更新实际上同时发生，且只在时钟上升开始下一个周期时。之所以能保持这样的等价性，是由于 Y86 指令集的本质，

也是由于我们按照遵循以下原则的方式来组织计算的：

"处理器从来不需要为了完成一条指令的执行而去读出该指令更新的状态。"

这条原则对我们实现的成功来说至关重要。

为了说明问题，假设我们实现 pushl 指令是先将%esp 减 4，再将更新后的%esp 值作为写操作的地址。这种方法就同前面所说的那个原则相违背。为了执行存储器操作，它需要先从寄存器文件中读更新过的栈指针。而我们的实现（图 4.18）产生出减过了的栈指针值，作为信号 valE，然后再用这个信号既作为寄存器写的数据，也作为存储器写的地址。因此，在时钟上升开始下一个周期时，处理器就可以同时执行寄存器写和存储器写了。

再举个例子来说明一下这条原则，我们可以看到有些指令（整数运算）会设置条件码，有些指令（跳转指令）会读取条件码，但没有指令必须既设置又读取条件码。虽然要到时钟上升开始下一个周期时，才会设置条件码，但是在任何指令试图读之前，它们都会更新好的。

下面这段代码是汇编代码，左边列出的是指令地址，图 4.23 给出了 SEQ 硬件是如何处理其中第 3 和第 4 行指令的：

```
1    0x000:    irmovl $0x100,%ebx          # %ebx <-- 0x100
2    0x006:    irmovl $0x200,%edx          # %edx <-- 0x200
3    0x00c:    addl %edx,%ebx              # %ebx <-- 0x300 CC <-- 000
4    0x00e:    je dest                     # Not taken
5    0x013:    rmmovl %ebx,0(%edx)         # M[0x200] <-- 0x300
6    0x019: dest: halt
```

标号为 1～4 的各个图给出了四个状态元素，还有组合逻辑，以及状态元素之间的连接。组合逻辑被条件码寄存器环绕着，因为有的组合逻辑（例如 ALU）产生输入到条件码寄存器，而其他部分（例如分支计算和 PC 选择逻辑）又将条件码寄存器作为输入。图中寄存器文件和数据存储器有分离的读连接和写连接，因为读操作沿着这些单元传播，就好像它们是组合逻辑，而写操作是由时钟控制的。

图 4.23 中的代码表明电路信号是如何与正在被执行的不同指令相联系的。我们假设处理是从设置条件码开始的，按照 ZF、SF 和 OF 的顺序，设为 100。在时钟周期 3 开始的时候（点 1），状态元素保持的是第二条 irmovl 指令（第二行）更新过的状态，该指令用中度灰色表示。组合逻辑用白色表示，表明它还没有来得及对变化了的状态做出反应。时钟周期开始时，地址 0x00c 载入程序计数器中。这样就会取出和处理用浅灰色表示的 addl 指令（第三行）。值沿着组合逻辑流动，包括读随机访问存储器。在这个周期末尾（点 2），组合逻辑为条件码产生了新的值（000），更新了程序寄存器%ebx，以及程序计数器的新值（0x00e）。在此时，组合逻辑已经根据 addl 指令（用浅灰色表示）被更新了，但是状态还是保持着第二条 irmovl 指令（用中度灰色表示）设置的值。

当时钟上升开始周期 4 时（点 3），会更新程序计数器、寄存器文件和条件码寄存器，因此我们用浅灰色来表示，但是组合逻辑还没有对这些变化做出反应，所以用白色表示。在这个周期内，会取出并执行 je 指令（第四行），在图中用深灰色表示。因为条件码 ZF 为 0，所以不会选择分支。在这个周期末尾（点 4），程序计数器已经产生了新值 0x00e。组合逻辑已经根据 je 指令（用深灰色表示）被更新过了，但是直到下个周期开始，状态还是保持着 addl 指令（用浅灰色表示）设置的值。

如此例所示，用时钟来控制状态元素的更新，以及值通过组合逻辑来传播，足够控制我们 SEQ 实现中每条指令执行的计算了。每次时钟由低变高时，处理器开始执行一条新指令。

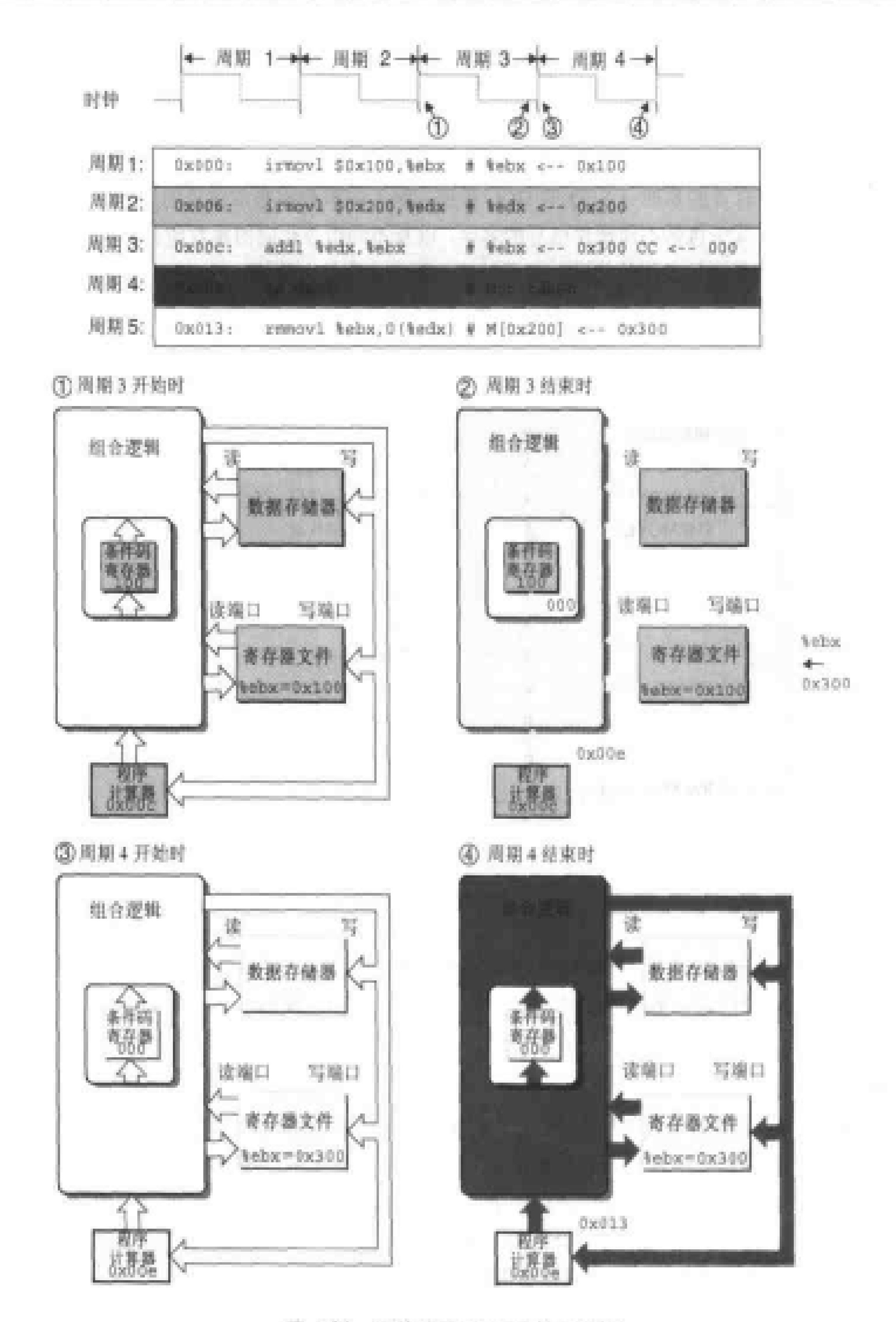


图 4.23 跟踪 SEQ 的两个执行周期

每个周期开始时，根据前一条指令设置状态元素（程序计数器、条件码寄存器、寄存器文件以及数据存储器）。信号传播到组合逻辑时，创建出新的状态元素的值。在下一个周期开始时，这些值会被加载到状态元素中。

4.3.4 SEQ的阶段实现

在本节中，我们会设计实现SEQ所需要的控制逻辑块的HCL描述。SEQ的所有HCL描述请参见附录A的A.2部分。在此，我们给出一些例子，而其他的只是作为练习题。我们建议你用这些练习来检验你的理解，即这些块是如何与不同指令的计算需求相联系的。

我们在这儿没有讲的那部分SEQ的HCL描述，是不同整数和布尔信号的定义，它们可以作为HCL操作的参数。其中包括不同硬件信号的名字，以及不同指令代码的常数值、寄存器名字和ALU操作。图4.24列出了我们使用的常数。按照习惯，常数值都是大写的。

名 称	值（十六进制）	含 义
INOP	0	nop 指令的代码
IHALT	1	halt 指令的代码
IRRMOVL	2	rrmovl 指令的代码
IIRMOVL	3	irmovl 指令的代码
IRMMOVL	4	rmmovl 指令的代码
IMRMOVL	5	mrmovl 指令的代码
IOPL	6	整数运算指令的代码
IJXX	7	跳转指令的代码
ICALL	8	call 指令的代码
IRET	9	ret 指令的代码
IPUSHL	a	pushl 指令的代码
IPOPL	b	popl 指令的代码
REST	6	%esp 的寄存器 ID
RNONE	8	表明没有寄存器文件访问
ALUADD	0	加法运算的功能

图 4.24 HCL描述中使用的常数值

这些值描述的是指令的编码、寄存器ID以及ALU操作。

除了图4.16～图4.19中所示的指令以外，我们还包括了对nop和halt指令的处理。这两条指令都是简单地经过各个阶段，不进行任何处理，除了要将PC加1。我们不会介绍halt指令实际上如何停止处理器的细节。只是简单假设当遇到icode为1时，处理器就停下来。

取指阶段

如图4.25所示，取指阶段包括指令存储器硬件单元。以PC作为第一个字节（字节0）的地址，这个单元一次从存储器读出六个字节。第一个字节被当成指令字节，（被标号为“Split”的单元）分为两个四位的量icode和ifun。根据icode的值，我们可以计算三个一位的信号（用虚线表示）：

instr_valid：这个字节对应于一个合法的Y86指令吗？这个信号用来发现不合法的指令。

need_regids：这个指令包括一个寄存器指示符字节吗？

need_valC：这个指令包括一个常数字吗？

让我们再来看一个例子，need_regids的HCL描述只是确定了icode的值是否是一条带有寄存器指示值字节的指令。

```
bool need_regids =
```

```
icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
           IIRMOVL, IRMMOVL, IMRMOVL };
```

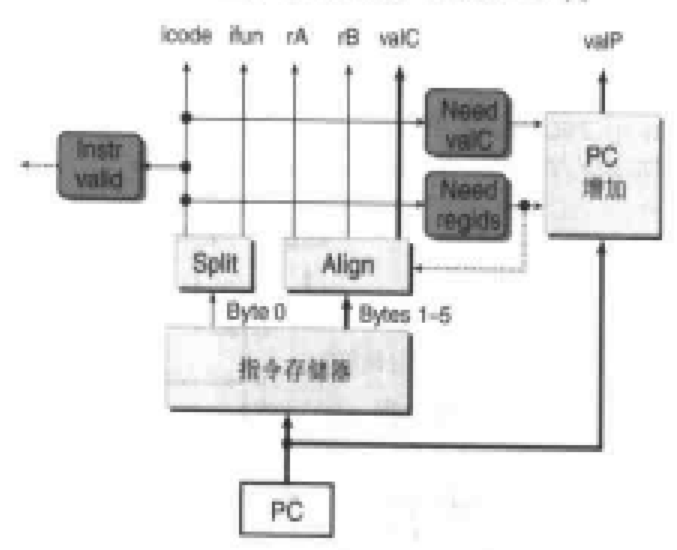


图 4.25 SEQ 取指阶段

以 PC 作为起始地址，从指令存储器中读出六个字节。根据这些字节，我们产生出各个指令字段。PC 增加块计算信号 valP。

练习题 4.14

写出 SEQ 实现中信号 need_valC 的 HCL 代码。

如图 4.25 所示，从指令存储器中读出的剩下五个字节是寄存器指示符字节和常数字的组合编码。标号为"Align"的硬件单元会处理这些字节，将它们放入寄存器字段和常数字中。当被计算的信号 need_regids 为 1 时，字节 1 被分开装入寄存器指示符 rA 和 rB 中。否则，这两个字段会被设为 8（RNONE），表明这条指令没有指明寄存器。回想一下（图 4.2），任何只有一个寄存器操作数的指令，寄存器指示值字节的另一个字段都设为 8（RNONE）。因此，我们可以将信号 rA 和 rB 看成，要么放着我们想要访问的寄存器，要么表明不需要访问任何寄存器。标号为"Align"的单元还产生常数字 valC。根据信号 need_regids 的值，要么根据字节 1~4 来产生 valC，要么根据字节 2~5 来产生。

PC 增加器（incrementer）硬件单元根据当前的 PC 以及两个信号 need_regids 和 need_valC 的值，产生信号 valP。对于 PC 值 p、need_regids 值 r 以及 need_valC 值 i，增加器产生值 $p+r+4i$。

解码和写回阶段

图 4.26 给出了 SEQ 中实现解码和写回阶段的逻辑的详细情况。这两个阶段联系在一起是因为它们都要访问寄存器文件。

寄存器文件有四个端口，它支持同时进行两个读（在端口 A 和 B 上）和两个写（在端口 E 和 M 上）。每个端口都有地址连接和数据连接，地址连接是一个寄存器 ID，而数据连接是一组 32 根线路，既可以作为寄存器文件的输出字（对读端口来说），也可以作为它的输入字（对写端口来说）。两个读端口的地址输入为 srcA 和 srcB，而两个写端口的地址输入为 dstA 和 dstB。如果某个地址端口上的值为特殊标识符 8（RNONE），则表明不需要访问寄存器。

根据指令代码 icode 以及寄存器指示值 rA 和 rB，图 4.26 底部的四个块产生出四个不同的寄存器文件的寄存器 ID。寄存器 ID srcA 表明应该读哪个寄存器以产生 valA。所需要的值是依赖于指令

类型的，如图 4.16~图 4.19 中解码阶段第一行中所示。将所有这些条目都整合到一个计算中就得到下面的 srcA 的 HCL 描述（回想一下 RESP 是%esp 的寄存器 ID）：

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

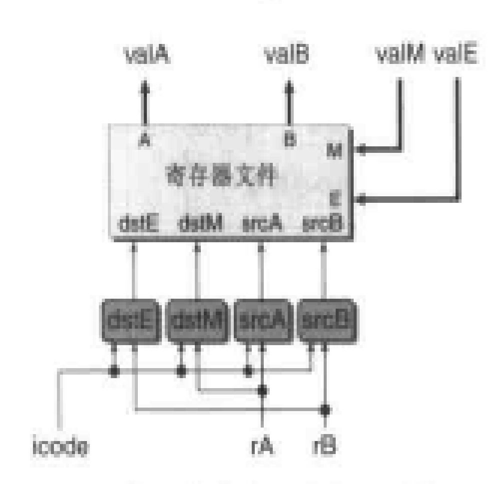


图 4.26 SEQ 解码和写回阶段

指令字段解码，用来产生寄存器文件使用的四个地址（两个读和两个写）的寄存器标识符。从寄存器文件中读出的值成为信号 valA 和 valB。两个写回值 valE 和 valM 作为写操作的数据。

练习题 4.15

寄存器信号 srcB 表明应该读哪个寄存器以产生 valB。所需要的值如图 4.16~图 4.19 中解码阶段第二行中所示。写出 srcB 的 HCL 代码。

寄存器 ID dstE 表明写端口 E 的目的寄存器，计算出来的值 valE 将放在那里，如图 4.16~4.19 中写回阶段第一个步骤所示。综合所有不同指令的目的寄存器，就得到下面的 dstE 的 HCL 描述：

```
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

练习题 4.16

寄存器 ID dstM 表明写端口 M 的目的寄存器，从存储器中读出来的值 valM 将放在那里，如图 4.16~图 4.19 中写回阶段第二个步骤所示。写出 dstM 的 HCL 代码。

练习题 4.17

只有 popl 指令会同时用到寄存器文件的两个写端口。对于指令 popl %esp，E 和 M 两个写端口会用到同一个地址，但是写入的数据不同。为了解决这个冲突，我们必须对两个写端口设立一个优先级，这样一来，当同一个周期内两个写端口都试图对一个寄存器进行写时，只有较高优先级端口上的写才会发生。那么为了实现练习题 4.5 中确定的行为，哪个端口该具有较高的优先级呢？

执行阶段

执行阶段包括算术/逻辑单元（ALU）。这个单元根据 alufun 信号的设置，对输入 aluA 和 aluB 执行 ADD、SUBTRACT、AND 或 EXCLUSIVE-OR 运算。如图 4.27 所示，这些数据和控制信号是由三个控制块产生的。ALU 的输出就是 valE 信号。

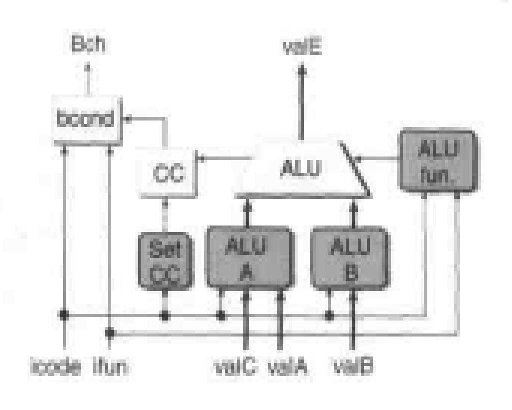


图 4.27 SEQ 执行阶段

ALU 要么为整数运算指令执行操作，要么作为加法器。根据 ALU 的值，设置条件码寄存器。检测条件码的值，判断是否该选择分支。

在图 4.16～图 4.19 中，执行阶段的第一个步骤给出的就是每条指令的 ALU 计算。列出的操作数 aluB 在前面，后面是 aluA，这样是为了保证 subl 指令是 valA 减去 valB。我们可以看到，根据指令的类型，aluA 的值可以是 valA、valC，或者是-4 或+4。因此我们可以用下面的方式来表达产生 aluA 的控制块的行为：

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

练习题 4.18

根据图 4.16～图 4.19 中执行阶段第一步的第一个操作数，写出 SEQ 中信号 aluB 的 HCL 描述。

观察 ALU 在执行阶段执行的操作，我们可以看到它通常是作为加法器来使用的。不过，对于 OPl 指令，我们希望它使用指令 ifun 字段中编码的操作。因此，我们可以将 ALU 控制的 HCL 描述写成：

```
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;
];
```

执行阶段还包括条件码寄存器。每次运行时，我们的 ALU 都会产生三个与条件码相关的信号——零、符号和溢出。不过，我们只希望在执行 OPl 指令时才设置条件码。因此我们产生了一个信号 set_cc 来控制是否该更新条件码寄存器：

```
bool set_cc = icode in { IOPL };
```

标号为"bcond"的硬件单元会确定一条指令是将导致跳转（选择分支），还是会继续下一条指令（不选择分支），并产生信号 Bch。只有当指令是一条跳转指令（icode 等于 IJXX），并且条件码的值和跳转类型（编码在 ifun 中）表明要选择分支（参见图 3.11）时，一条指令才会导致跳转。我们将省略这个单元的设计。

访存阶段

存储器阶段的任务就是读或者写程序数据。如图 4.28 所示，两个控制块产生存储器地址和存储器输入数据（为写操作）的值。另外两个块产生表明应该执行读操作还是写操作的控制信号。当执

行读操作时，数据存储器产生值 valM。

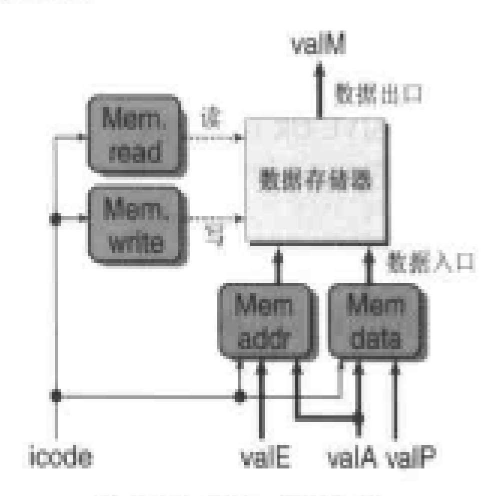


图 4.28 SEQ 访存阶段

数据存储器既可以写，也可以读存储器的值。从存储器中读出的值就形成了信号 valM。

每个指令类型所需要的存储器操作在图 4.16~图 4.19 的存储器阶段中给出来了。可以看到存储器读和写的地址总是 valE 或 valA。这个块用 HCL 描述就是：

```
int mem_addr = [
icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
icode in { IPOPL, IRET } : valA;
# Other instructions don't need address
];
```

练习题 4.19

观察图 4.16~图 4.19 所示的不同指令的存储器操作，我们可以看到存储器写的数据总是 valA 或 valP。写出 SEQ 中信号 mem_data 的 HCL 代码。

我们希望只为从存储器读数据的指令设置控制信号 mem_read，用 HCL 代码表示就是：

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

练习题 4.20

我们希望只为向存储器写数据的指令设置控制信号 mem_write。写出 SEQ 中信号 mem_write 的 HCL 代码。

更新 PC 阶段

SEQ 中最后一个阶段会产生程序计数器的新值（见图 4.29）。如图 4.16~图 4.19 中最后步骤所示，取决于指令的类型和是否要选择分支，新的 PC 可能是 valC、valM 或 valP。用 HCL 来描述这个选择就是：

```
int new_pc = [
        # Call. Use instruction constant
        icode == ICALL : valC;
        # Taken branch. Use instruction constant
        icode == IJXX && Bch : valC;
```

```
    # Completion of RET instruction. Use value from stack
    icode == IRET : valM;
    # Default: Use incremented PC
    1 : valP;
];
```

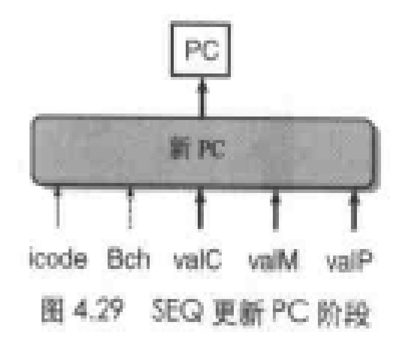


图 4.29 SEQ 更新 PC 阶段

根据指令代码和分支标志，从信号 valC、valM 和 valP 中选出下一个 PC 的值。

SEQ 小结

现在我们已经浏览过 Y86 处理器的一个完整的设计。我们可以看到，通过将执行每条不同指令所需的步骤组织成一个统一的流程，就可以用很少量的各种硬件单元以及一个时钟来控制计算的顺序，从而实现整个处理器。不过这样一来，控制逻辑就必须要在这些单元之间路由信号，并根据指令类型和分支条件产生适当的控制信号。

SEQ 惟一的问题就是它太慢了。时钟必须非常慢，以使信号能在一个周期内传播过所有的阶段。让我们来看看处理一条 ret 指令的例子。在时钟周期起始时，从更新过的 PC 开始，要从指令存储器中读出指令，从寄存器文件中读出栈指针，ALU 要减小栈指针，为了得到程序计数器的下一个值，还要从存储器中读出返回地址。所有这一切都必须在这个周期结束之前完成。

这种实现方法不能充分利用硬件单元，因为每个单元只在整个时钟周期的一部分时间内才被使用。我们会看到引入流水线能获得更好的性能。

4.3.5 SEQ+：重新安排计算阶段

作为到流水线化的设计的一个中间步骤，我们将重新排列这六个阶段的顺序，使得更新 PC 阶段在一个周期开始时执行，而不是结束时才执行，这样产生的处理器设计称为 SEQ+，因为它扩展了基本的 SEQ 处理器。这种做法看上去有些奇怪，因为确定新的 PC 值需要检测执行阶段中的分支条件（对条件转移来说），或者读访存阶段中的返回值（对 ret 指令来说）。

如图 4.30 所示，我们能移动 PC 阶段，使得它的逻辑在时钟开始时活动，计算当前指令的 PC 值。然后这个 PC 值就可以输入到取指阶段，剩下的处理就和前面讲过的一样继续下去。在时钟周期结束之前，组合逻辑会产生计算新的 PC 值所需要的所有的信号。这些值放在一组寄存器中，在图中是用标号为“pState”（代表“previous state”）的方框来表示的。现在 PC 阶段的任务变成了为当前指令选择 PC 值，而不是为下一条指令计算更新了的 PC。

图 4.31 给出了 SEQ+硬件的一个更为详细的说明。我们可以看到，它包括与我们在 SEQ 中用到的（图 4.21）一样的硬件单元和控制块，只不过 PC 逻辑移到了底部。从前面一条指令得到的结果存放在图中底部所示的寄存器中，它们的标号是它们所保存的值前面加上一个前缀字母“p”（代表“previous”）。

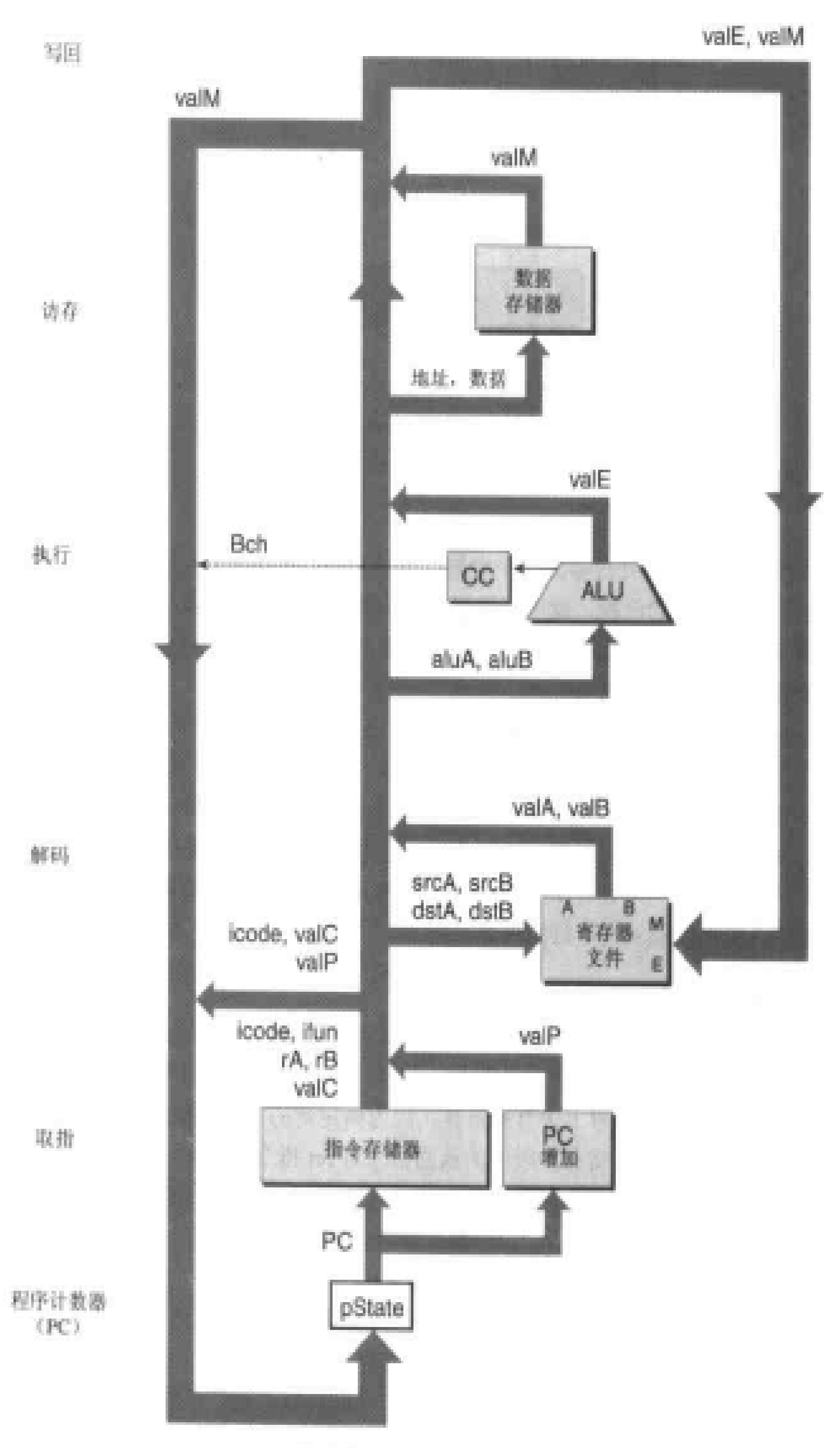


图 4.30 SEQ+的抽象视图

在这个版本中，当前指令的程序计数器（PC）的选择是根据前一个周期的信息，在一个时钟周期开始时计算的。这种结构能帮助我们得到一个流水线化的实现。

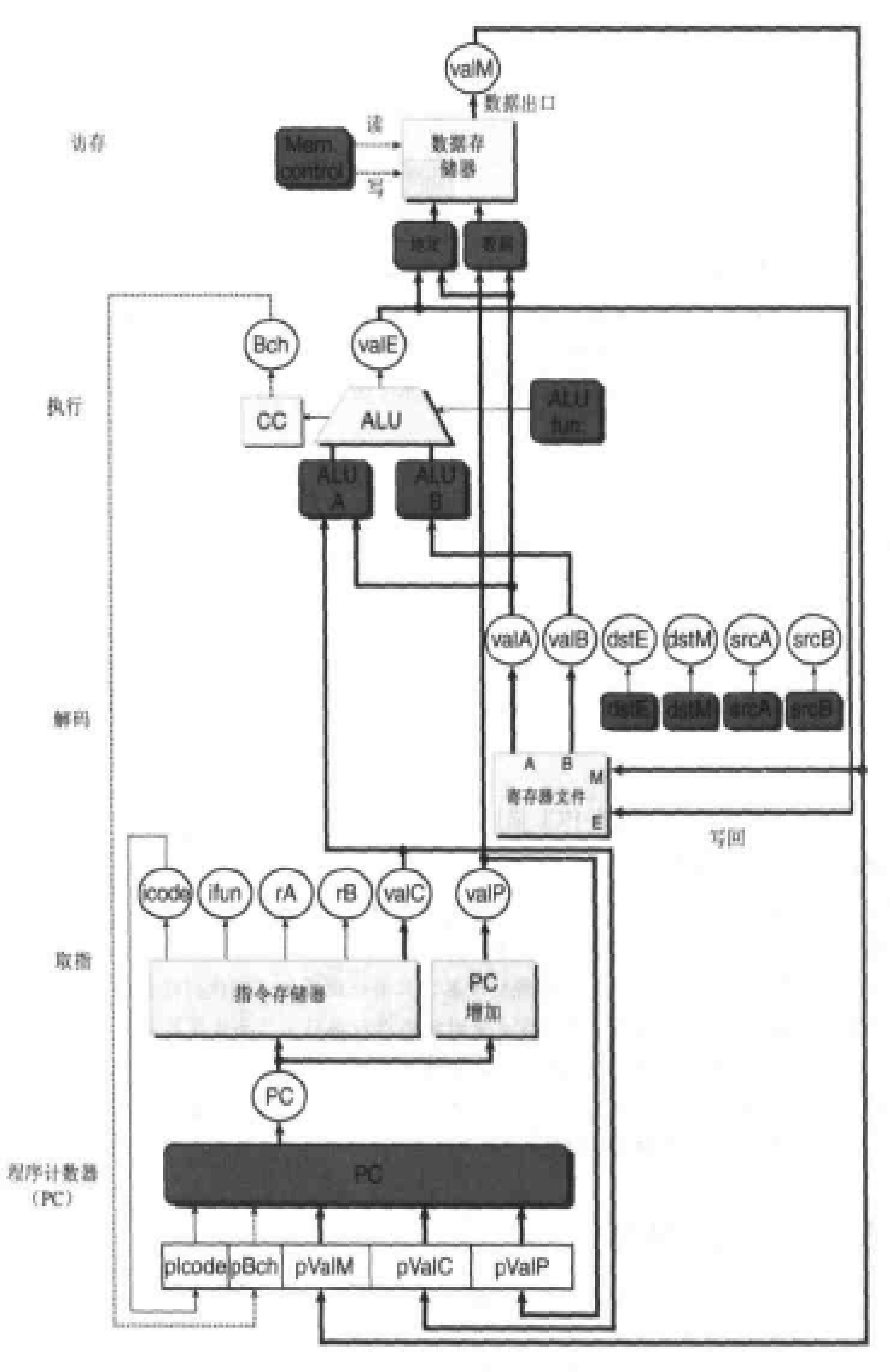
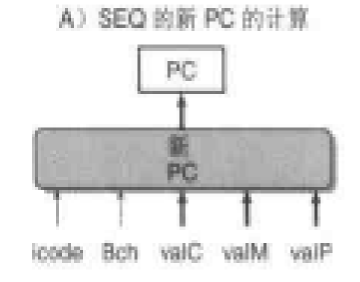


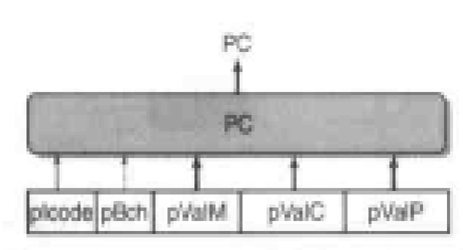
图 4.31 SEQ+的硬件结构

图中省略了一些信号。

对控制逻辑的惟一修改就是重新定义了 PC 的计算，使它使用以前的状态值。下面这两个图表

明了 SEQ 和 SEQ+的 PC 计算块：





我们看到，两个块之间惟一的区别就是，将保存着处理器状态的寄存器从 PC 计算的后面移到了前面。这个例子是一种很常见的改进，称为**电路重定时**（circuit retiming）。重定时改变了一个系统的状态表示，但是并不改变它的逻辑行为。通常用来平衡一个系统中各个部分之间的延迟。

PC 计算的 HCL 描述变成了：

```
int pc = [
        # Call. Use instruction constant
        pIcode == ICALL : pValC;
        # Taken branch. Use instruction constant
        pIcode == IJXX && pBch : pValC;
        # Completion of RET instruction. Use value from stack
        pIcode == IRET : pValM;
        # Default: Use incremented PC
        1 : pValP;
];
```

附录 A 的 A.3 节是 SEQ+的所有 HCL 描述。

旁注：SEQ+中的 PC 在哪里？

SEQ+有一个很奇怪的特色，那就是没有硬件寄存器来存放程序计数器。相反，是根据从前一条指令保存下来的一些状态信息来动态地计算 PC 的。这就是一个小小的例证，证明我们可以以一种与 ISA 隐含着的概念模型不同的方式来实现处理器，只要处理器能正确执行任意的机器语言程序。我们不需要按照程序员可见的状态表明的方式来对状态进行编码，只要处理器能对任意程序员可见的状态（例如，程序计数器）产生正确的值。在创建流水线化的设计中，我们会更多地使用到这条原则。5.7 节中描述的乱序（out-of-order）处理技术，以一种完全不同于机器级程序中发生顺序的次序来执行指令，将这一思想发挥到了极致。

4.4 流水线的通用原理

在试图设计一个流水线化的 Y86 处理器之前，让我们先来看看流水线化的系统的一些通用属性和原理。对于曾经在自助餐厅的服务线上工作过或者开车通过自动汽车清洗线的人，都会非常熟悉这种系统。在流水线化的系统中，待执行的任务被划分成了若干个独立的阶段。在自助餐厅，这些阶段包括提供沙拉、主菜、甜点以及饮料。在汽车清洗中，这些阶段包括喷水和打肥皂、擦洗、上蜡和烘干。通常都会允许多个顾客同时经过系统，而不是要等到一个用户完成了所有从头至尾的过程才让下一个开始。在一个典型的自助餐厅流水线上，顾客按照相同的顺序经过各个阶段，即使他

们并不需要某些菜。对汽车清洗来说，当前面一辆汽车从喷水阶段进入擦洗阶段时，下一辆就可以进入喷水阶段了。通常，汽车必须以相同的速度通过这个系统，避免撞车。

流水线化的一个重要特性就是增加了系统的吞吐量（throughput），也就是单位时间内服务的顾客总数，不过它也会轻微地增加执行时间（latency），也就是服务一个用户需要的时间。例如，自助餐厅里的一个只需要沙拉的顾客，能很快通过一个非流水线化的系统，只在沙拉阶段稍做停留。但是在流水线化的系统中，这个顾客如果试图直接去沙拉阶段就有可能招致其他顾客的愤怒了。

4.4.1 计算流水线

让我们把注意力放到计算流水线上来，这里的“顾客”就是指令，每个阶段执行指令的一部分。图 4.32 给出了一个很简单的非流水线化的硬件系统例子。它是由一些执行计算的逻辑以及一个保存计算结果的寄存器组成的。时钟信号控制在每个特定的时间间隔加载寄存器。CD 播放器中的解码器就是这样的一个系统。输入信号是从 CD 表面读出的位，逻辑部分对这些位进行解码，产生音频信号。图中的计算块是用组合逻辑来实现的，意味着信号会穿过一系列逻辑门，在一定时间的延迟之后，输出就成为了输入的某个函数。



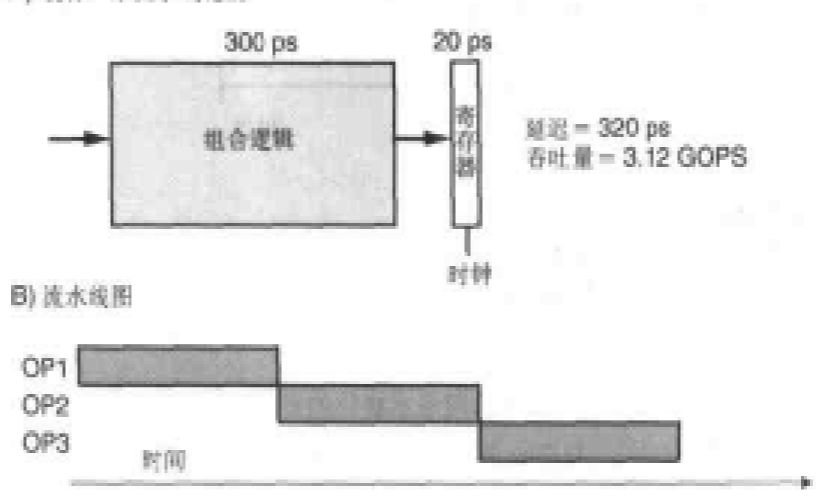


图 4.32 非流水线化的计算硬件

每个 320ps 的周期内，系统用 300ps 计算组合逻辑函数，20ps 将结果存到输入寄存器中。

在时序逻辑设计中，电路延迟是以微微秒（picosecond，简写成“ps”），也就是 10^{-12} 秒，来计算的。在这个例子中，我们假设组合逻辑需要 300ps，而加载寄存器需要 20ps。图 4.32 还给出了一种时序图，称为流水线图（pipeline diagram）。在图中，时间从左向右流动。从上到下写着一组操作（在此称为 OP1、OP2 和 OP3）。实心的长方形表示执行这些操作的时间。这个系统中，在开始下一个操作之前必须完成前一个。因此，这些方框在垂直方向上并没有相互重叠。下面这个公式给出了运行这个系统的最大吞吐量：

$$\text{Thoughput}=\frac{1\text{operation}}{(20+300)\text{picosecond}}\cdot\frac{1000\text{picosecond}}{1\text{nanosecond}}\approx 3.12\text{GOPS}$$

我们以每秒千兆次操作（简写成 GOPS），也就是每秒十亿次操作，为单位来描述吞吐量。从头到尾执行一条指令所需要的时间称为执行时间（latency）。在此系统中，执行时间为 320ps，也就是

吞吐量的倒数。

假设将我们的系统执行的计算分成三个阶段（A、B 和 C），每个阶段需要 100ps，如图 4.33 所示。然后在各个阶段之间放上流水线寄存器（pipeline registers），这样每个操作都会按照三步经过这个系统，从头到尾需要三个完整的时钟周期。如图 4.33 中的流水线图所示，只要 OP1 从 A 进入 B，就可以让 OP2 进入阶段 A 了，依此类推。在稳定状态下，三个阶段都应该是活动的，每个时钟周期，一个操作离开系统，一个新的进入。从流水线图中第三个时钟周期就能看出这一点，此时，OP1 是在阶段 C，OP2 在阶段 B，而 OP3 是在阶段 A。在这个系统中，我们将时钟周期设为 100+20=120ps，得到的吞吐量大约为 8.33GOPS。因为处理一条操作需要 3 个时钟周期，所以这条流水线的执行时间就是 3×120=360ps。我们将系统吞吐量提高到原来的 8.33/3.12=2.67 倍，代价是增加了一些硬件，以及执行时间的少量增加（360/320=1.12）。执行时间变大是由于增加的流水线寄存器的时间开销。

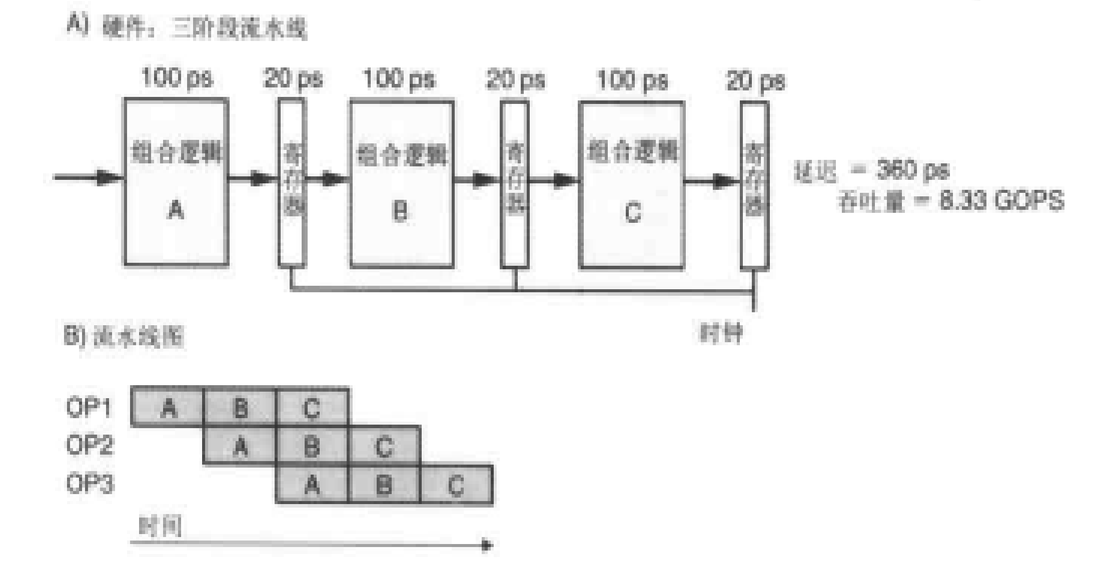


图 4.33　三阶段流水线化的计算硬件

计算被划分为三个阶段 A、B 和 C。每经过一个 120ps 的周期，每个操作就行进通过一个阶段。

4.4.2　流水线操作的详细说明

为了更好地理解流水线是怎样工作的，让我们来详细看看流水线计算的时序和操作。图 4.34 给出了前面我们看到过的三阶段流水线（图 4.33）的流水线图。就像流水线图上方表明的那样，流水线阶段之间的操作转移是由时钟信号来控制的。每隔 120ps，信号从 0 升至 1，开始流水线阶段的下一组计算。

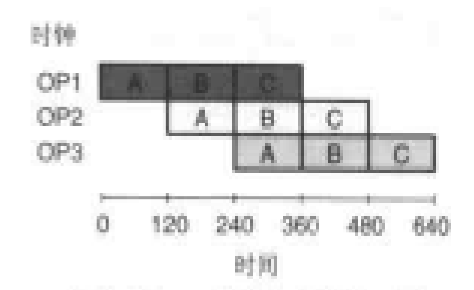


图 4.34　三阶段流水线的时序

时钟信号的上升沿控制操作从一个流水线阶段移动到下一个阶段。

之前，阶段A中计算的操作OP2的值已经到达第一个流水线寄存器的输入，但是该寄存器的状态和输出还保持为操作OP1在阶段A中计算的值。操作OP1在阶段B中计算的值已经到达第二个流水线寄存器的输入。当时钟上升时，这些输入被加载进流水线寄存器，成为寄存器的输出（点2）。另外，阶段A的输入被设置成开始操作OP3的计算。然后信号传播通过各个阶段的组合逻辑（点3）。就像图中点3处的曲线化的波阵面（curved wavefront）表明的那样，信号可能以不同的速率通过各个不同的部分。在时刻360之前，结果值到达流水线寄存器的输入（点4）。当时刻360时钟上升时，各个操作会前进经过一个流水线阶段。

从这个对流水线操作详细的描述中，我们可以看到减缓时钟操作不会影响流水线的行为。信号传播到流水线寄存器的输入，但是直到时钟上升时才会改变寄存器的状态。另一方面，如果时钟运行得太快，就会有灾难性的后果。值可能会来不及通过组合逻辑，因此当时钟上升时，寄存器的输入还不是合法的值。

根据我们对SEQ处理器的时序的讨论（4.3.3节），我们看到那种在组合逻辑块之间采用时钟寄存器的简单机制就足够控制流水线中的操作流了。随着时钟周而复始的上升和下降，不同的操作就会通过流水线的各个阶段，不会相互干扰。

4.4.3 流水线的局限性

图4.33的例子给出了一个理想的流水线化的系统，在这个系统中，我们可以将计算分成三个相互独立的阶段，每个阶段需要的时间是原来逻辑需要时间的三分之一。不幸的是，还有其他一些因素会减弱流水线的效率。

不一致的划分

图4.36展示的系统中，和前面一样，我们将计算划分为了三个阶段，但是通过这些阶段的延迟从50ps到150ps不等。通过所有阶段的延迟和仍然为300ps。不过，我们运行时钟的速率是由最慢的阶段的速度限制的。正如本图中流水线图表明的那样，每个时钟周期，阶段A都会空闲（用白色方框表示）100ps，而阶段C会空闲50ps。只有阶段B会一直处于活动状态。我们必须将时钟周期设为150+20=170ps，也就是吞吐量为5.88 GOPS。另外，由于时钟周期减慢了，执行时间也增加到了510ps。

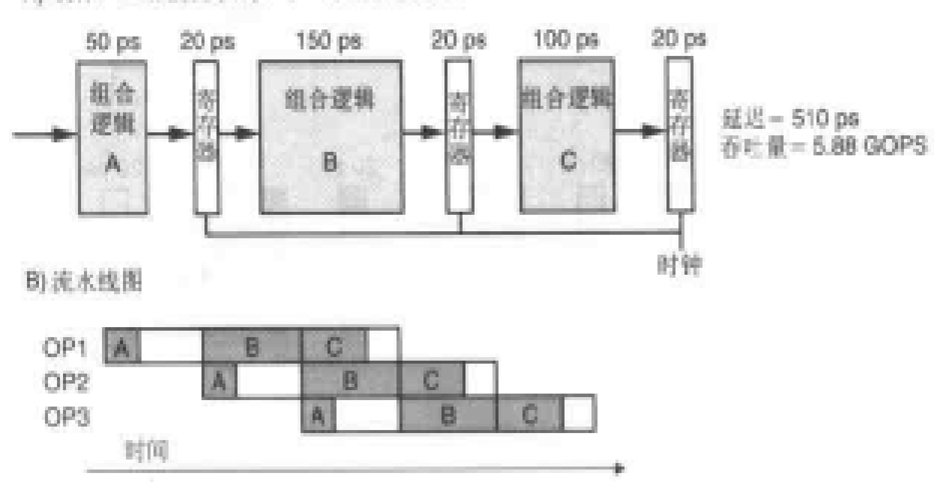


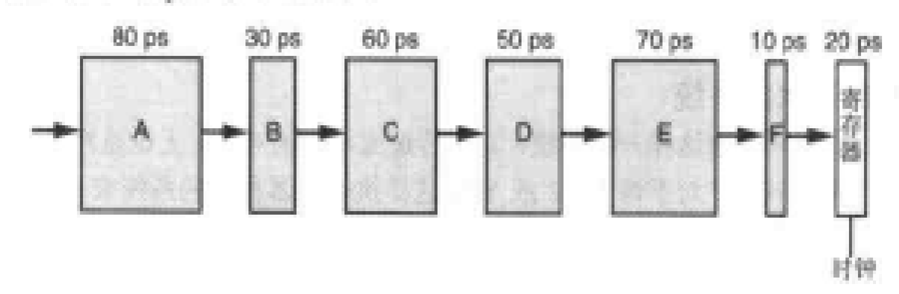
图4.36 由不一致的阶段延迟造成的流水线技术的局限性

系统的吞吐量为最慢阶段的速度所限。

对硬件设计者来说，将系统计算设计划分成一组具有相同延迟的阶段是一个主要的挑战。通常，处理器中的某些硬件单元，如 ALU 和存储器，是不能被划分成多个延迟较小的单元的，这就使得创建一组平衡的阶段非常困难。在设计我们的流水线化的 Y86 处理器中，我们不会过于关注这一层次的细节，但是理解时序优化在实际系统设计中的重要性还是非常重要的。

练习题 4.21

假设我们分析图 4.32 中的组合逻辑，认为它可以分成 6 个块，依次命名为 A ~ F，延迟分别为 80、30、60、50、70 和 10ps，如下图所示：



在这些块之间插入流水线寄存器，就得到这一设计的流水线化的版本。根据在哪里插入流水线寄存器，会出现不同的流水线深度（有多少个阶段）和最大吞吐量的组合。假设每个流水线寄存器的延迟为 20ps。

A. 只插入一个寄存器，得到一个两阶段的流水线，要使吞吐量最大化，该在哪里插入寄存器呢？吞吐量和执行时间是多少？

B. 要使一个三阶段的流水线的吞吐量最大化，该将两个寄存器插在哪里呢？吞吐量和执行时间是多少？

C. 要使一个四阶段的流水线的吞吐量最大化，该将三个寄存器插在哪里呢？吞吐量和执行时间是多少？

D. 要得到一个吞吐量最大的设计，至少要有几个阶段？描述这个设计及其吞吐量和执行时间。

流水线过深，收益反而下降

图 4.37 说明了流水线技术的另一个局限性。在这个例子中，我们把计算分成了 6 个阶段，每个阶段需要 50ps。在每对阶段之间插入流水线寄存器就得到了一个六阶段流水线。这个系统的最小时钟周期为 50+20=70ps，吞吐量为 14.29 GOPS。因此，通过将流水线的阶段数加倍，我们将性能提高了 14.29/8.33=1.71。虽然我们将每个计算时钟的时间缩短了两倍，但是由于通过流水线寄存器的延迟，吞吐量并没有加倍。这个延迟成了流水线吞吐量的一个制约因素。在我们的新设计中，这个延迟占到了整个时钟周期的 28.6%。

现代处理器为了提高时钟频率，采用了很深的（15 或更多的阶段）流水线。处理器设计师将指令的执行划分成很多非常简单的步骤，这样一来每个阶段的延迟就很小。电路设计者小心地设计流水线寄存器，使其延迟尽可能得小。芯片设计者也必须小心地设计时钟传播网络，以保证时钟在整个芯片上同时改变。所有这些都是设计高速微处理器面临的挑战。

练习题 4.22

让我们来看看图 4.32 中的系统，将它划分成任意多个流水线阶段，每个阶段有相同的延迟。如果给定流水线寄存器的延迟为 20ps，吞吐量的上限是多少呢？

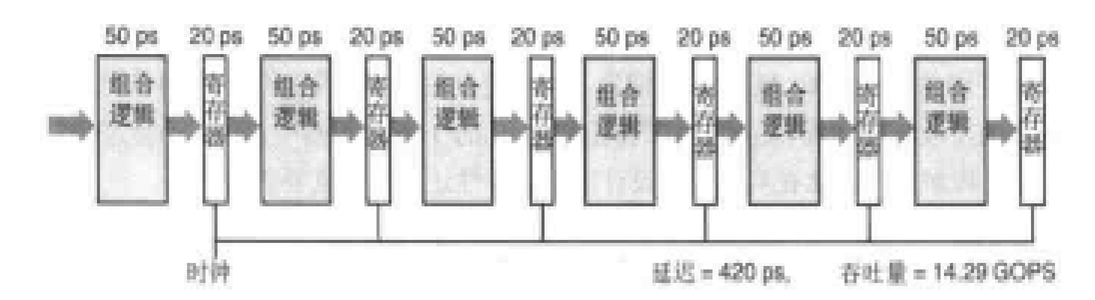
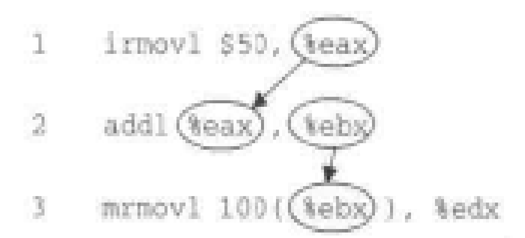


图 4.37 由开销造成的流水线技术的局限性

在组合逻辑被分成较小的块时，由寄存器更新引起的延迟就成为了一个限制因素。

4.4.4 带反馈的流水线系统

到目前为止，我们只考虑这样一种系统，其中传过流水线的对象，无论是汽车、人，还是指令，相互都是完全独立的。但是，对于像 IA32 或 Y86 这样执行机器程序的系统来说，相邻指令之间很可能是相关的。例如，考虑下面这个 Y86 指令序列：



在这个包含三条指令的序列中，每对相邻的指令之间都有数据相关（data dependency），用带圈的寄存器名字和它们之间的箭头来表示。irmovl 指令（第一行）将它的结果存放在%eax 中，然后 addl 指令（第二行）要读这个值；而 addl 指令将它的结果存放在%ebx 中，mrmovl 指令（第三行）要读这个值。

另一种相关是由指令控制流造成的顺序相关（sequential dependency）。来看看下面这个 Y86 指令序列：

```
1   loop:
2       subl %edx,%ebx
3       jne targ
4       irmovl $10,%edx
5       jmp loop
6   targ:
7       halt
```

jne 指令（第三行）产生了一个控制相关（control dependency），因为条件测试的结果会决定要执行的新指令是 irmovl 指令（第四行）还是 halt 指令（第七行）。在我们的 SEQ 设计中，这些相关都是由反馈路径来解决的，如图 4.20 的右边所示。这些反馈将更新的寄存器值向下传送到寄存器文件，将新的 PC 值传送到 PC 寄存器。

图 4.38 举例说明了将流水线引入含有反馈路径的系统中的危险。在原来的系统（A）中，每个操作的结果都反馈给下一个操作。流水线图（B）就说明了这个情况，OP1 的结果成为 OP2 的输入，依此类推。如果我们试图将它转换成一个三阶段流水线（C），我们将改变系统的行为。如

流水线图（C）所示，OP1 的结果成为 OP4 的输入。为了通过流水线技术加速系统，我们改变了系统的行为。

当我们将流水线技术引入 Y86 处理器时，我们必须正确处理反馈的影响。很明显，像图 4.38 中例子那样改变系统的行为是不可接收的。我们必须以某种方式来处理指令间的数据和控制相关，以使得到的行为与 ISA 定义的模型相符。

A）硬件：未流水线化，带反馈

组合逻辑

寄存器

时钟

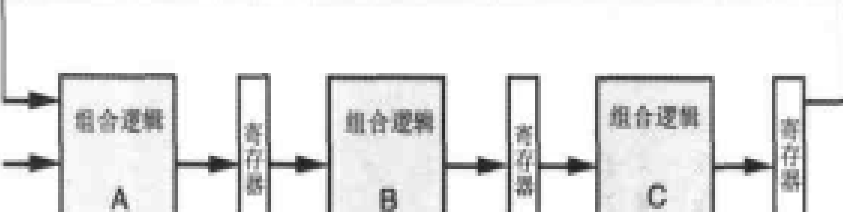
B）流水线图

OP1

OP2

OP3

时间



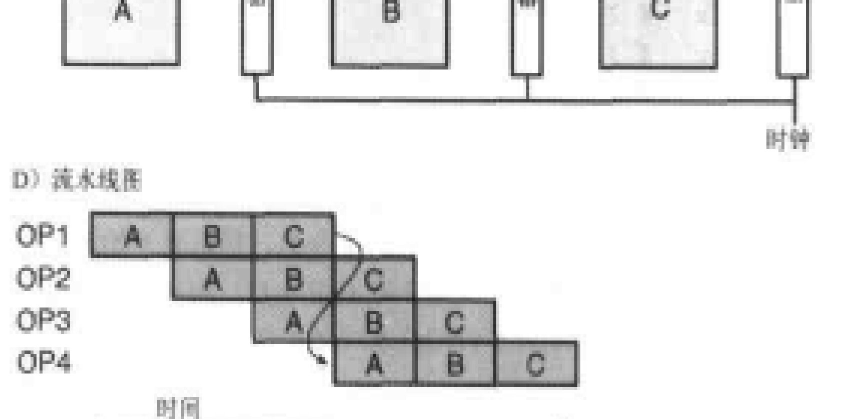


图 4.38　由逻辑相关造成的流水线技术的局限性

在从未流水线化的带反馈的系统（A）转化到流水线化的系统（C）的过程中，我们改变了它的组合逻辑的行为，可以从两个流水线图（B 和 D）中看出来。

4.5 Y86的流水线实现

我们终于准备好要开始本章的主要任务——设计一个流水线化的Y86处理器了。我们开始时以SEQ+作为基础，在各个阶段之间添加流水线寄存器。我们最初并不尝试正确地处理不同的数据和控制相关。不过，经过一些修改，我们将达到目的，得到一个实现Y86 ISA的、有效的、流水线化的处理器。

4.5.1 插入流水线寄存器

在我们创建一个流水线化的Y86处理器的最初尝试中，我们要在SEQ+的各个阶段之间插入流水线寄存器，并对信号重新做了排列，得到PIPE-处理器，这里名字中的“-”代表这个处理器和最终的处理器设计相比，性能要差一点。PIPE-的抽象结构如图4.39所示。流水线寄存器在该图中用浅灰色方框表示。每个寄存器可以存放多个字节和字，待会我们会看到。可以观察到PIPE-使用的硬件单元与我们的两个顺序设计：SEQ（图4.20）和SEQ+（图4.30）完全一样。

流水线寄存器是按如下方式标号的：

F 保存程序计数器的预测值，待会儿会讨论。

D 位于取指和解码阶段之间。它保存关于最新取出的指令的信息，即将由解码阶段进行处理。

E 位于解码和执行阶段之间。它保存关于最新解码的指令和从寄存器文件读出的值的信息，即将由执行阶段进行处理。

M 位于执行和访存阶段之间。它保存最新执行的指令的结果，即将由访存阶段进行处理。它还保存关于用于处理条件转移的分支条件和分支目标的信息。

W 位于访存阶段和反馈路径之间，反馈路径将计算出来的值提供给寄存器文件写，而当完成ret指令时，它还要向PC选择逻辑提供返回地址。

图4.40表明的是下面这段代码序列是怎样通过我们的五阶段流水线的，其中对各条指令的注释用I1～I5来表示：

```
1    irmovl  $1,%eax   # I1
2    irmovl  $2,%ecx   # I2
3    irmovl  $3,%edx   # I3
4    irmovl  $4,%ebx   # I4
5    halt              # I5
```

图中右边给出了这个指令序列的流水线图。同4.4节中简单流水线化的计算单元的流水线图一样，这个图描述了每条指令通过流水线各个阶段的行进过程，时间是从左往右增大的。上面一条数字表明各个阶段发生的时钟周期。例如，在周期1取出指令I1，然后它开始通过流水线各个阶段，到周期5结束时，其结果写入寄存器文件。在周期2取出指令I1，到周期6结束时，其结果写回，以此类推。在最下面，我们给出了当周期为5时的流水线的扩展图。此时，每个流水线阶段中各有一条指令。

从图4.40中，我们还可以看到我们画处理器的习惯是合理的，这样，指令是自底向上的流动的。周期5时的扩展图表明的流水线阶段，取指阶段在底部，写回阶段在最上面，正如流水线硬件图（图4.39）表明的一样。如果看看流水线各个阶段中指令的顺序，就会发现它们出现的顺序与在程序中列出的顺序一样。因为正常的程序是从上到下列出的，我们保留这种顺序，让流水线从下到上进行。

在使用与本内容有关的模拟器时，这个习惯会特别有用。

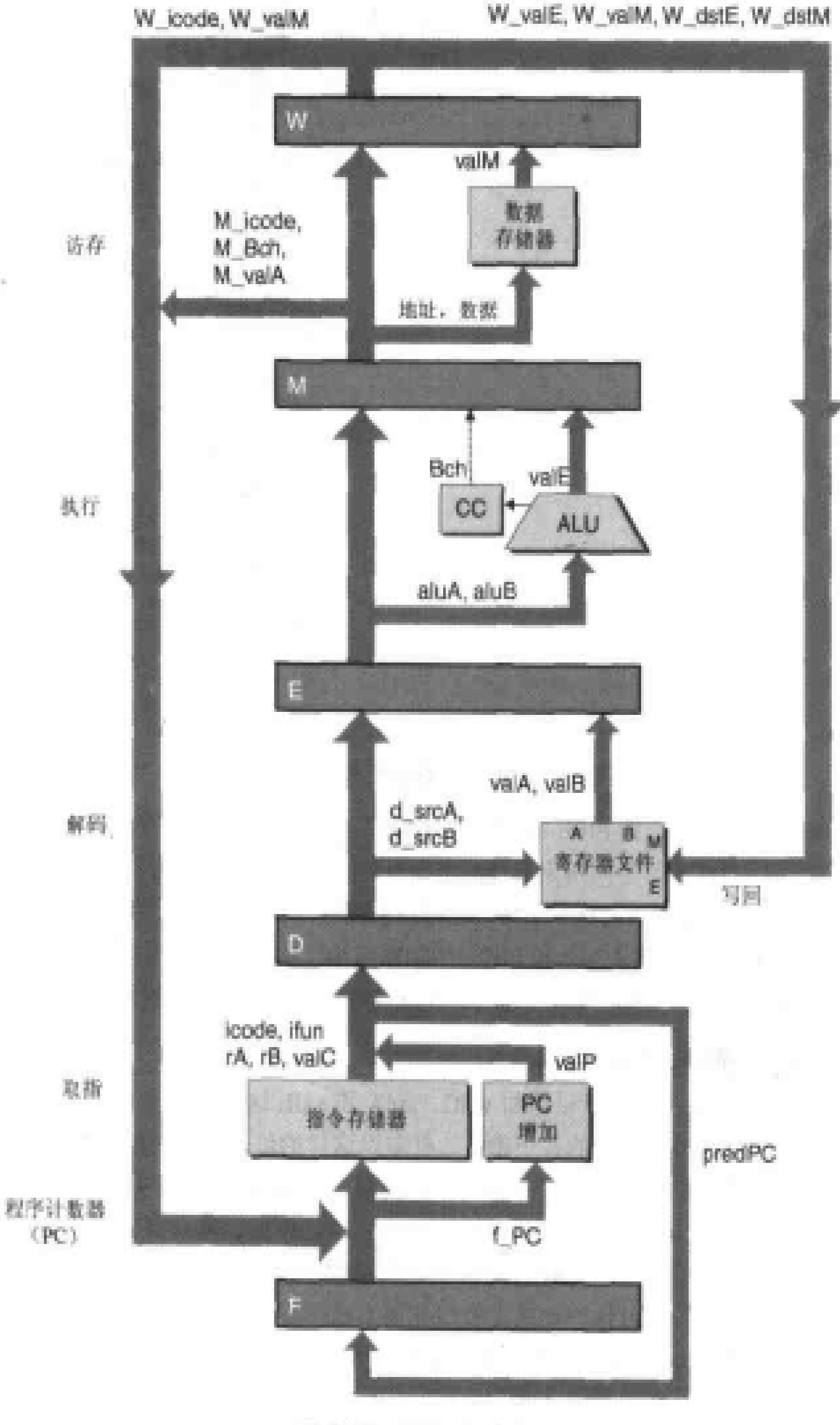


图 4.39　PIPE-的抽象视图

通过往 SEQ+（图 4.30）中插入流水线寄存器，我们创建了一个五阶段的流水线。这个版本有几个缺陷，待会儿我们就会解决这些问题。

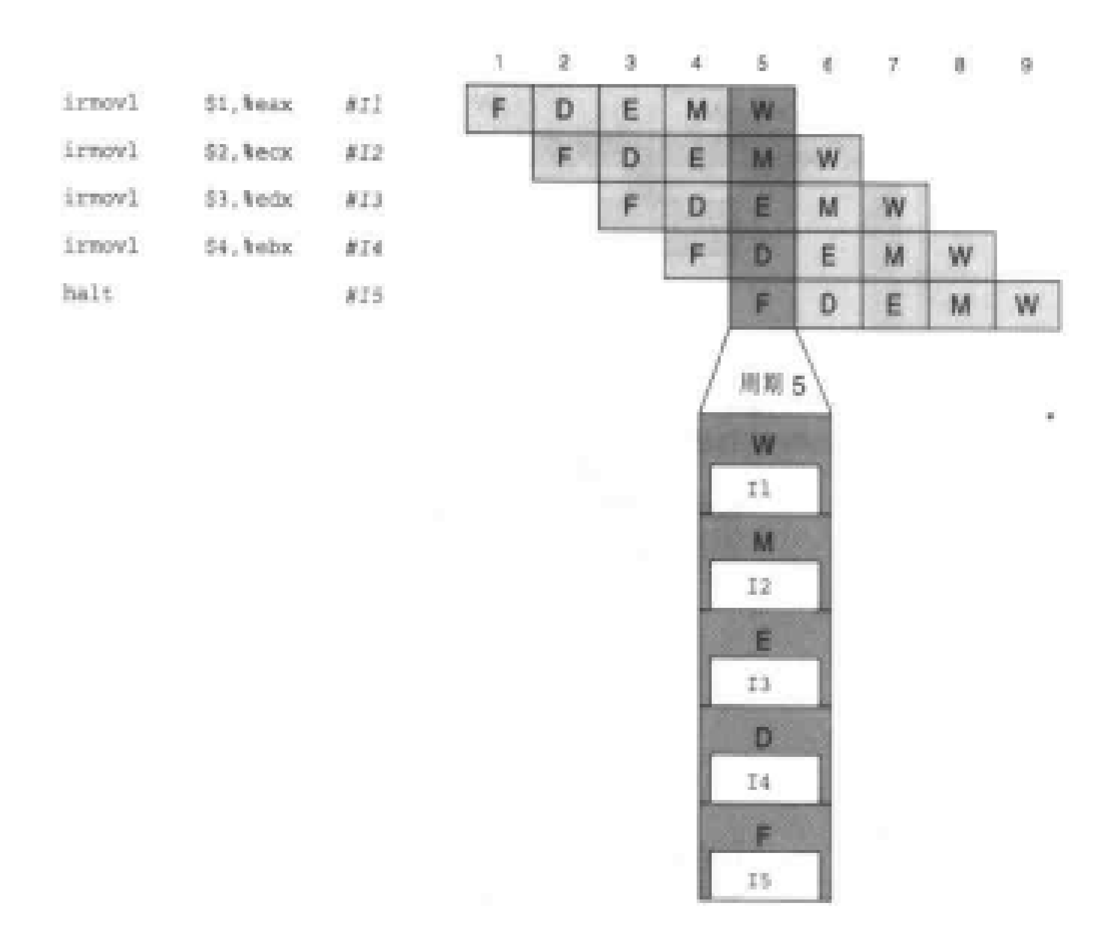


图 4.40 指令流通过流水线的示例

图 4.41 给出了一个更详细的 PIPE-硬件结构的说明。可以看到每个流水线寄存器包含多个字段（用白色方框表示），对应于与不同指令通过流水线有关的信号。与两个顺序处理器的硬件结构（图 4.21 和 4.31）中用圆角方框表示的标号不同，这些白色方框代表实际的硬件部件。

比较 SEQ+的抽象结构（图 4.30）和 PIPE-的抽象结构（图 4.39），我们看到虽然通过各个阶段的流非常相似，但是还是有一些细微的差别的。在继续详细实现之前，让我们来看看这些差别。

4.5.2 对信号进行重新排列和标号

SEQ+一次只处理一条指令，所以诸如 valC、srcA 和 valE 这样的信号有惟一的值。在流水线化的设计中，对应于正在经过系统的各个指令，有多组这样的值。例如，在 PIPE-的详细结构中，有四个标号为“icode”的白色方框，保存着四个不同指令的 icode 信号（见图 4.41）。我们要很小心地保证使用的是正确的信号值，否则就会出现严重错误，例如，将一条指令计算出的结果存入另一条指令指定的目的寄存器中。在我们采用的命名机制中，通过在信号名字前加上大写的流水线寄存器的名字作为前缀，可以惟一地确定存放在流水线寄存器中的信号。例如，四个 icode 值分别命名为 D_icode、E_icode、M_icode 和 W_icode。我们还需要引用某些在一个阶段内刚刚计算出来的信号。它们的命名是在信号名前面加上小写的阶段名的第一个字母作为前缀，例如 d_srcA 和 e_Bch。

SEQ+和 PIPE-的解码阶段都产生信号 dstE 和 dstM，它们指明值 valE 和 valM 的目的寄存器。

在 SEQ+中，我们可以将这些信号直接连到寄存器文件写端口的地址输入。在 PIPE-中，会在流水线中一直携带这些信号穿过执行和访存阶段，直到写回阶段才送到寄存器文件（见各个阶段的详细描述所示）。我们这样做是为了确保写端口的地址和数据输入是来自同一条指令。否则，会将处于写回阶段的指令的值写入，而寄存器 ID 却来自于处于解码阶段的指令。作为一条通用原则，我们要保存处于一个流水线阶段中的指令的所有信息。

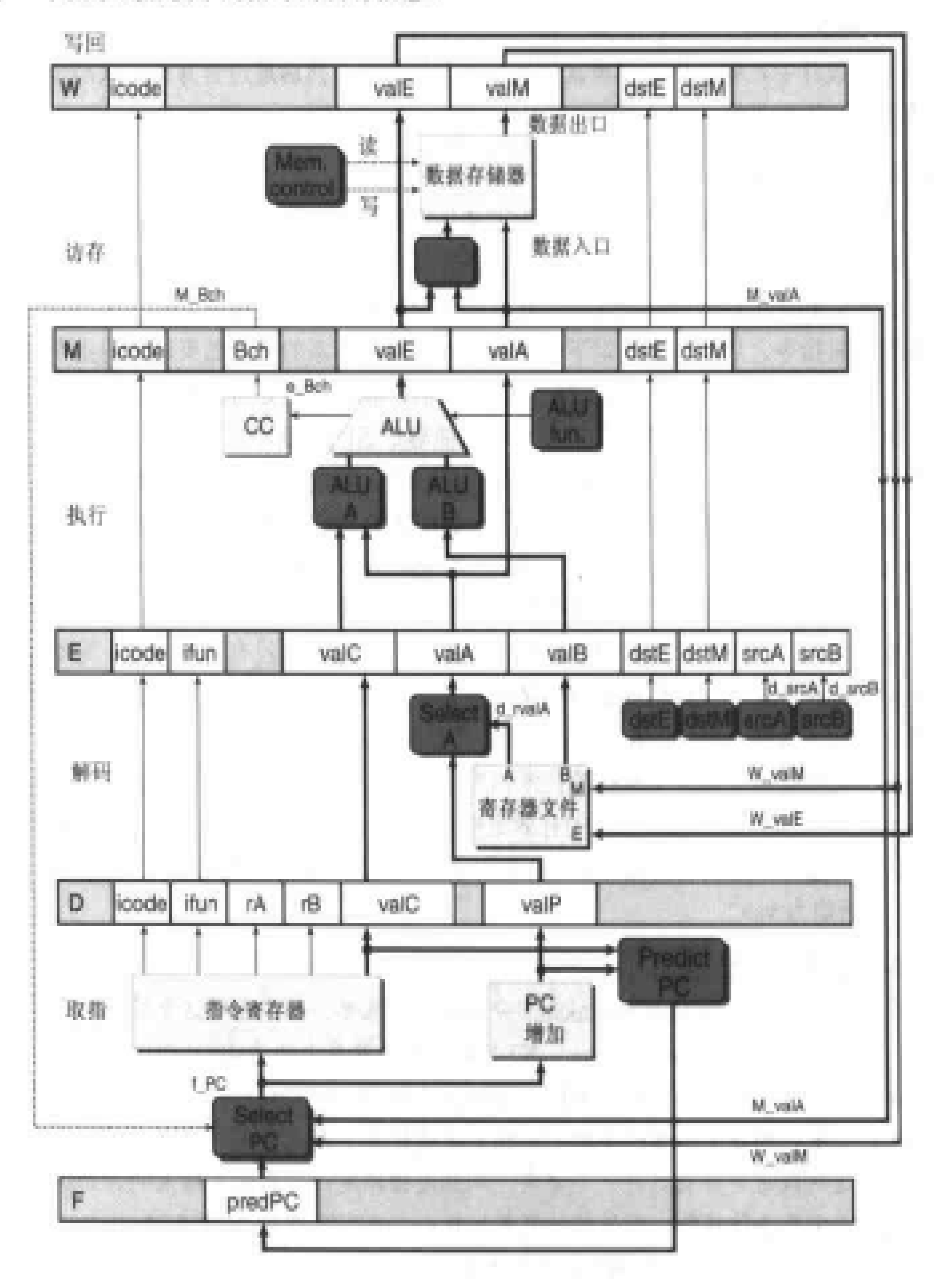


图 4.41 PIPE-的硬件结构，一个初始的流水线化的实现

并没有画出所有的连接。

PIPE-中有一个块是在相同表示形式的 SEQ+中没有的，那就是解码阶段中标号为“Select A”

的块。我们可以看出，这个块会从来自流水线寄存器 D 的 valP 或从寄存器文件 A 端口中读出的值中选择一个，作为流水线寄存器 E 的值 valA。包括这个块是为了减少要携带给流水线寄存器 E 和 M 的状态数量。在所有的指令中，只有 call 在访存阶段需要 valP 的值，只有跳转指令在执行阶段（当不需要进行跳转时）需要 valP 的值，而这些指令又都不需要从寄存器文件中读出的值。因此我们合并这两个信号，将它们作为信号 valA 携带穿过流水线，从而可以减少流水线寄存器的状态数量。这样做就消除了 SEQ（图 4.21）和 SEQ+（图 4.31）中标号为“Data”的块，这个块完成的就是类似的功能。在硬件设计中，像这样仔细确认信号是如何使用的，然后通过合并信号来减少寄存器状态和线路的数量，是很常见的。

4.5.3 预测下一个 PC

在 PIPE-设计中，我们采取了一些措施来正确处理控制相关。我们流水线化的设计的目的就是每个时钟周期都发射（issue）一条新指令，也就是说每个时钟周期都有一条新指令进入执行阶段并最终完成。要是达到这个目的也就意味着吞吐量是每个时钟周期一条指令。为了达到这个目的，我们必须在取出当前指令之后，马上确定下一条指令的位置。不幸的是，如果取出的指令是条件分支指令，要到几个周期后，也就是指令通过执行阶段之后，我们才能知道是否要选择分支。类似地，如果取出的指令是 ret，要到指令通过访存阶段，才能确定返回地址。

除了条件转移指令和 ret 这些例外，根据取指阶段中计算的信息，我们能够确定下一条指令的地址。对于 call 和 jmp（无条件转移）来说，下一条指令的地址是指令中的常数字 valC，而对于其他指令来说就是 valP。因此，通过预测 PC 的下一个值，在大多数情况下，我们能达到每个时钟周期发射一条新指令的目的。对大多数指令类型来说，我们的预测是完全可靠的。对条件转移来说，我们既可以预测选择了分支，那么新 PC 值应为 valC，也可以预测没有选择分支，那么新 PC 值应为 valP。无论在哪种情况中，我们都必须以某种方式来处理我们预测错误的情况，因为此时我们已经取出并部分执行了错误的指令。我们会在 4.5.9 节中再讨论这个问题。

猜测分支方向并根据猜测开始取指的技术称为**分支预测**。实际上所有的处理器都采用了某种形式的此类技术。对于预测是否选择分支的有效策略已经进行了广泛的研究[31]。有的系统花费了大量硬件来解决这个任务。在我们的设计中，只使用了简单的策略，那就是总是预测选择了条件分支，因而预测 PC 的新值为 valC。

旁注：其他的分支预测策略

我们的设计使用总是选择（always taken）分支的预测策略。研究表明这个策略的成功率大约为 60%[31]。反过来，从不选择（never taken，NT）策略的成功率大约为 40%。一种稍微复杂一点的，称为反向选择、正向不选择（backward taken，forward not-taken，BTFNT）的策略，当分支地址较低时就预测选择分支，而分支地址较高时，就预测不选择分支而选择下一条指令。这种策略的成功率大约为 65%。这种改进是源自这样一个事实，那就是循环是由后向分支结束的，而循环通常会执行多次。前向分支用于条件操作，而选择的可能性较小。在家庭作业 4.39 和 4.40 中，你可以修改 Y86 流水线处理器来实现 NT 和 BTFNT 分支预测策略。

不成功的分支预测对程序性能的影响将在 5.12 节中程序优化的上下文中讨论。

我们还剩预测 ret 指令的新 PC 值没有讨论。同条件转移不同，此时可能的返回值几乎是无限的，

因为返回地址是位于栈顶的字，其内容可以是任意的。在我们的设计中，我们不会试图对返回地址做任何预测。我们只是简单地暂停处理新指令，直到 ret 指令通过写回阶段。在 4.5.9 节中，我们将回过来讨论这部分的实现。

旁注：带堆栈的返回地址预测

对大多数程序来说，很容易预测返回值，因为过程调用和返回是成对出现的。大多数函数调用，会返回到调用后的那条指令。高性能处理器中运用了这个属性，在取指单元中放入一个硬件栈，它保存着过程调用指令产生的返回地址。每次执行过程调用指令时，都将其返回地址压入栈中。当取出一个返回指令时，就从栈中弹出最上面的值，作为预测的返回值。同分支预测一样，也必须提供在预测错误时的恢复机制，因为还是有调用和返回不匹配的时候。通常，这种预测是很可靠的。这个硬件栈对程序员来说是不可见的。

PIPE-的取指阶段，如图 4.41 底部所示，负责预测 PC 的下一个值，以及为指令的读取选择实际的 PC。我们可以看到，标号为"Predict PC（预测 PC）"的块会从 PC 增加器（incrementer）计算出的 valP 和取出的指令中得到的 valC 中进行选择。这个值会存放在流水线寄存器 F 中，作为程序计数器的预测值。标号为"Select PC（选择 PC）"的块类似于 SEQ+的 PC 选择阶段中标号为"PC"的块（图 4.31）。它从三个值中选择一个作为指令存储器的地址：预测的 PC，对于不选择分支的指令来说的 valP 的值（该指令到达流水线寄存器 M，并且 valP 值存储在寄存器 M_valA 中），或是当 ret 指令到达流水线寄存器 W（存储在 W_valM）时的返回地址的值。

当我们在 4.5.9 节完成流水线控制逻辑时，会返回来处理跳转和返回指令的。

4.5.4 流水线冒险（hazard）

我们的 PIPE-结构是创建一个流水线化的 Y86 处理器的好开端。不过，回忆一下我们在 4.4.4 节中的讨论，将流水线技术引入一个带反馈的系统会导致相邻指令间在发生相关时出现问题。在完成我们的设计之前，必须解决这个问题。这些相关有两种形式：①**数据相关**，下一条指令会用到这一条指令计算出的结果；②**控制相关**，一条指令要确定下一条指令的位置，例如在执行跳转、调用或返回指令时。这些相关可能会导致流水线产生计算错误，称为**冒险**。同相关一样，冒险也可以分为两类：**数据冒险**（data hazard）和**控制冒险**（control hazard）。在本节中，我们关心的是数据冒险。我们会将控制冒险作为整个流水线控制的一部分加以讨论（4.5.9 节）。

图 4.42 描述的是 PIPE-处理器处理称为 prog1 的指令序列的情况。这段代码将值 10 和 3 放入程序寄存器%edx 和%eax，执行三条 nop 指令，然后将寄存器%edx 加到%eax。我们重点关注两条 irmovl 指令和 addl 指令之间的数据相关可能造成的数据冒险。在图的右边，我们给出了这个指令序列的流水线图。在这个流水线图中突出显示了周期 6 和 7 的流水线阶段。流水线图的下面是周期 6 中写回活动和周期 7 中解码活动的详细说明。在周期 7 开始以后，两条 irmovl 都已经通过写回阶段，所以寄存器文件保存着更新过的%edx 和%eax 的值。因此，当 addl 指令在周期 7 经过解码阶段时，它是可以读到源操作数的正确值的。在此示例中，两条 irmovl 指令和 addl 指令之间的数据相关没有造成数据冒险。

我们看到 prog1 通过流水线并得到正确的结果，因为三条 nop 指令在有数据相关的指令之间创造了一些延迟。让我们来看看如果去掉这些 nop 指令会发生些什么。图 4.43 描述的是一个叫做 prog2

的程序的流水线流程，在两条产生寄存器%edx 和%eax 值的 irmovl 指令和以这两个寄存器作为操作数的 addl 指令之间有两条 nop 指令。在这种情况下，关键步骤发生在周期 6，此时 addl 指令从寄存器文件中读取它的操作数。该图底部给出的是这个周期内流水线活动的详细描述。第一个 irmovl 指令已经通过了写回阶段，因此程序寄存器%edx 已经在寄存器文件中更新过了。在该周期内，第二个 irmovl 指令处于写回阶段，因此对程序寄存器%eax 的写要到周期 7 开始，时钟上升时，才会发生。结果，会读出%eax 的错误值（在此我们假设所有的寄存器的初始值为 0），因为对该寄存器的写还未发生。很明显，我们的流水线还不能正确处理这样的冒险。

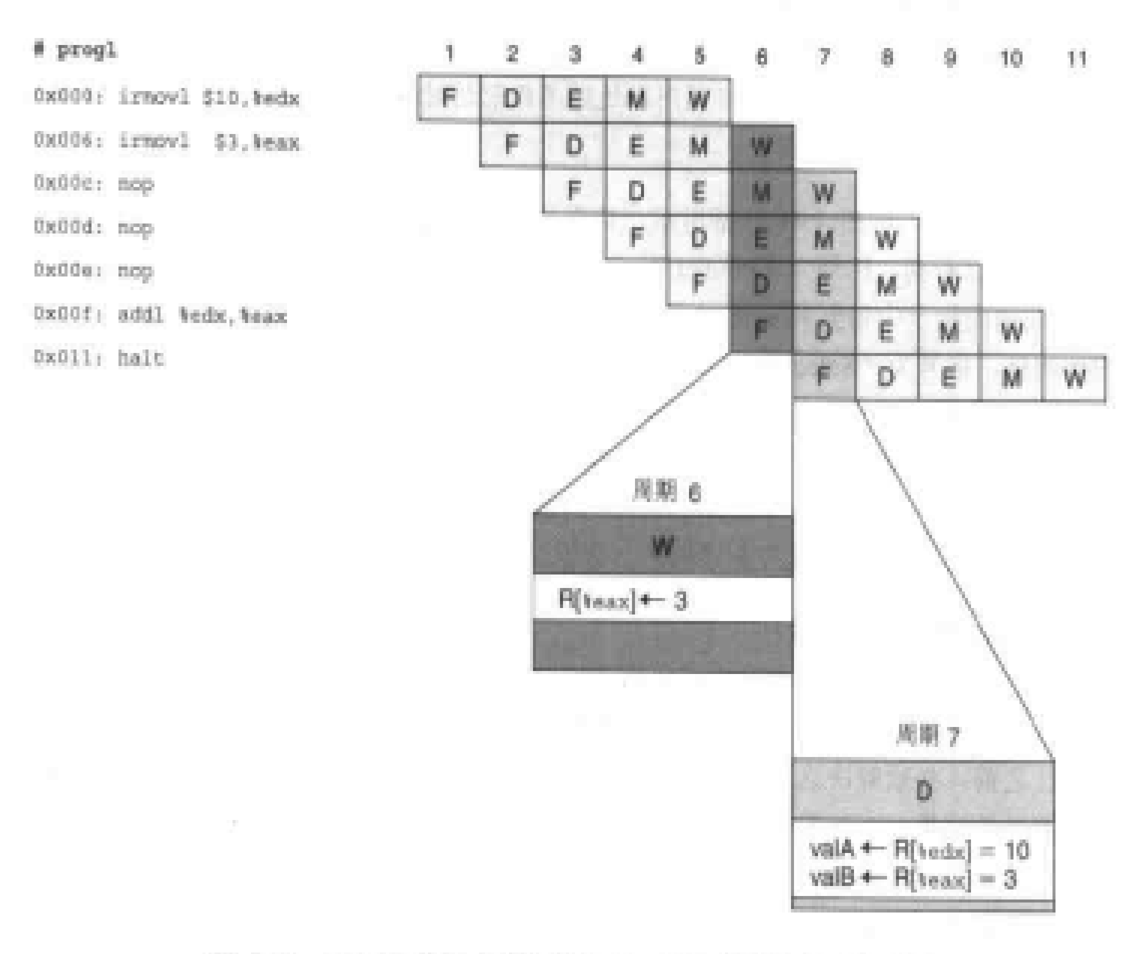


图 4.42 prog1 的流水线化的执行，没有特殊的流水线控制

在周期 6 中，第二个 irmovl 将结果写入寄存器%eax。addl 指令在周期 7 读源操作数，因此得到的是%edx 和%eax 的正确值。

图 4.44 给出的是当 rimovl 指令和 addl 指令之间只有一条 nop 指令，即为程序 prog3 时，发生的情况。现在我们必须检查周期 5 内流水线的行为，此时 addl 指令通过解码阶段。不幸的是，对寄存器%edx 的写仍处在写回阶段，而对寄存器%eax 的写还处在访存阶段。因此，addl 指令会得到两个错误的操作数。

图 4.45 给出的是当我们去掉 irmovl 指令和 addl 指令间的所有 nop 指令，即为程序 prog4 时，发生的情况。现在我们必须检查周期 4 内流水线的行为，此时 addl 指令通过解码阶段。不幸的是，对寄存器%edx 的写仍处在访存阶段，而执行阶段正在计算寄存器%eax 的新值。因此，addl 指令的两个操作数都是不正确的。

这些例子说明，如果一条指令的操作数被它前面三条指令中的任意一条改变的话，都会出现

数据冒险。之所以会出现这些冒险，是因为我们的流水线化的处理器是在解码阶段从寄存器文件中读取指令的操作数，而要到三个周期以后，指令经过写回阶段时，才会将指令的结果写到寄存器文件。

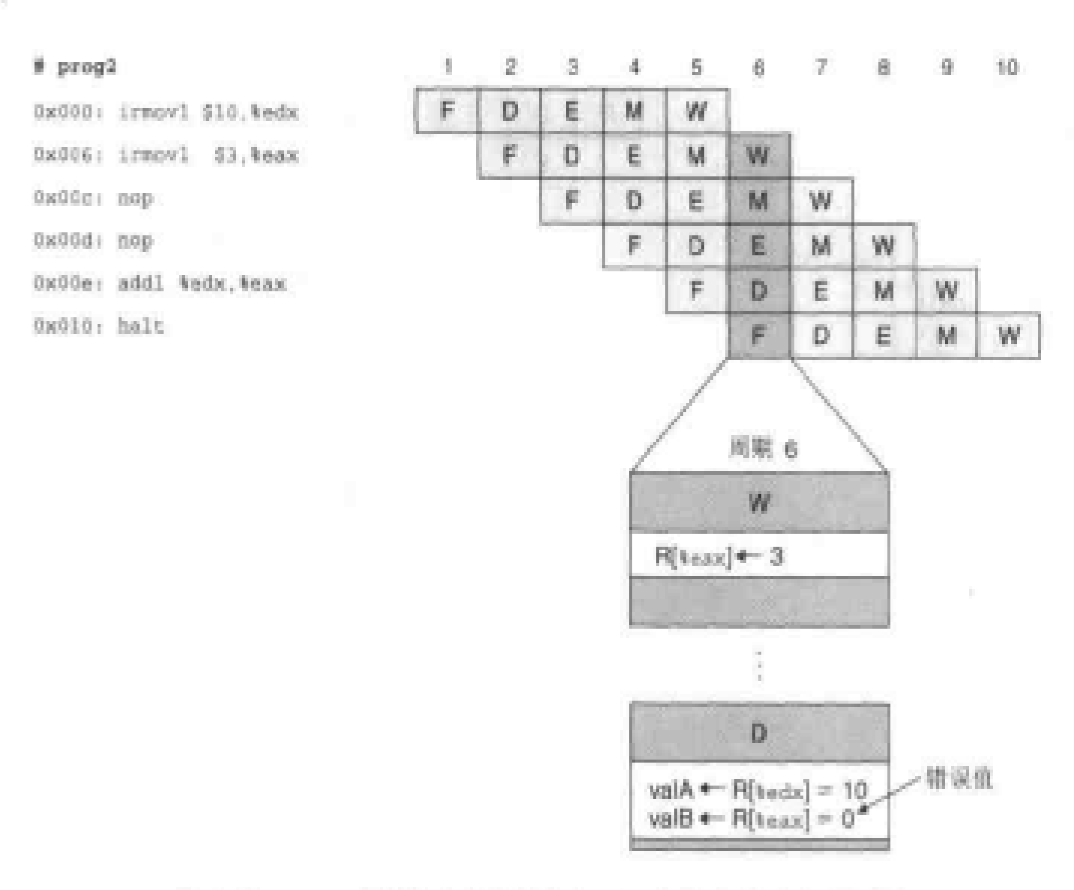


图 4.43 prog2 的流水线化的执行，没有特殊的流水线控制

直到周期 6 结束时，对寄存器%eax 的写才发生，导致 addl 指令在解码阶段读出的是该寄存器的错误值。

旁注：列举数据冒险的类型

当一条指令更新后面指令会读到的那些程序状态时，就有可能出现冒险。所谓的程序状态包括程序寄存器、条件码、存储器和程序计数器。下面让我们来看看每类状态出现冒险的可能性。

程序寄存器：我们已经认识这种冒险了。出现这种冒险是因为寄存器文件的读写是在不同的阶段进行的，导致不同指令之间可能出现不希望的相互作用。

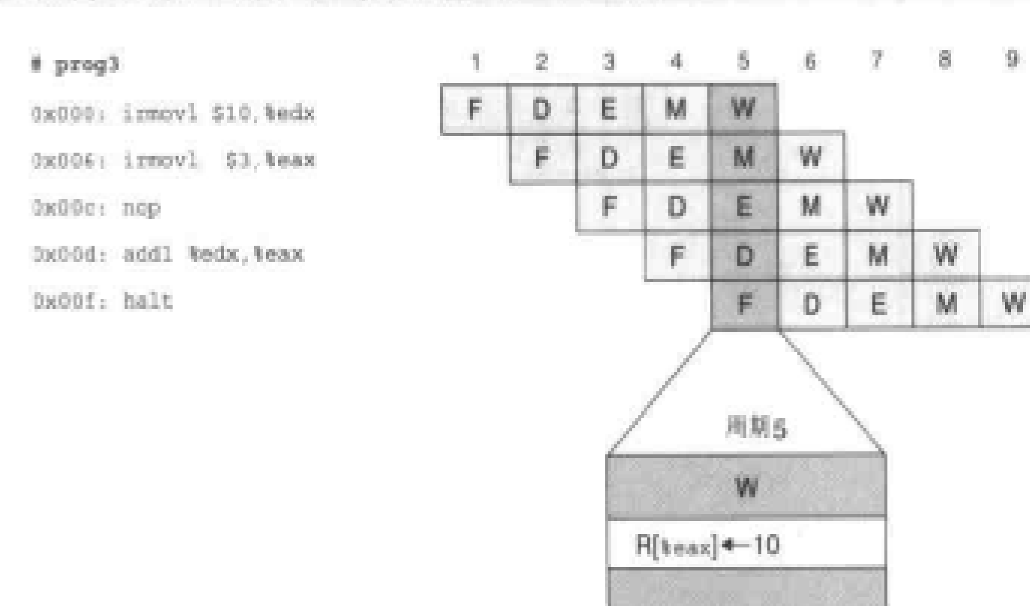
条件码：在执行阶段中，既可能写（整数操作）也可能读（条件转移）条件码。在条件转移经过这个阶段之前，前面所有的整数操作都已经完成这个阶段了。所以不会发生冒险。

程序计数器：更新和读取程序计数器之间的冲突导致了控制冒险。当我们的取指阶段逻辑在取下一条指令之前，正确预测了程序计数器的新值时，就不会产生冒险。预测错误的分支和 ret 指令需要特殊的处理，会在 4.5.9 节中讨论。

存储器：对数据存储器的读和写都发生在访存阶段。在一条读存储器的指令到达这个阶段之前，前面所有要写存储器的指令都已经完成这个阶段了。另外，在访存阶段中写数据的指令和在取指阶段中读指令之间也有冲突，因为指令和数据存储器访问的是同一个地址空间。只有包含自我修改代码（self-modifying code）的程序才会发生这种情况，在这样的程序中，指令写存储器的一部分，过

后会从中取出指令。有些系统有复杂的机制来发现和避免这种冒险，而有些系统只是简单地强制要求程序不应该使用自我修改代码。为了简便，我们假设程序不能修改自身。

这些分析表明我们只需要处理寄存器数据冒险和控制冒险。



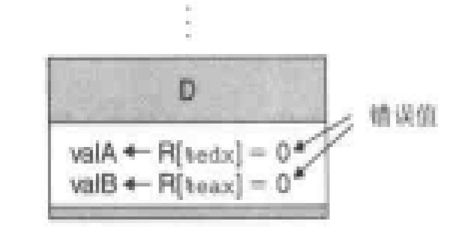


图 4.44 prog3 的流水线化的执行，没有特殊的流水线控制

在周期5，addl 指令从寄存器文件中读源操作数。对寄存器%edx 的写仍处在写回阶段，而对寄存器%eax 的写还在访存阶段。两个操作数 valA 和 valB 得到的都是错误值。

4.5.5 用暂停（stalling）来避免数据冒险

暂停（stalling）是一种常用的用来避免冒险的技术。暂停时，处理器会停止流水线中一条或多条指令，直到冒险条件不再满足。只要一条指令的源操作数会被流水线后面某个阶段中的指令产生，处理器就会通过将指令阻塞在解码阶段来避免数据冒险。图 4.46（prog2）、图 4.47（prog3）和图 4.48（prog4）就说明了这项技术。当指令 addl 处于解码阶段时，流水线控制逻辑发现执行、访存或写回阶段中至少有一条指令会更新寄存器%edx 或%eax。处理器不会让 addl 指令带着不正确的结果通过这个阶段，而是会暂停指令，将它阻塞在解码阶段，时间为一个周期（对 prog2 来说）、两个周期（对 prog3 来说）或者甚至于三个周期（对 prog4 来说）。对所有这三个程序来说，addl 指令最终都会在周期 7 中得到两个源操作数的正确值，然后继续沿着流水线进行下去。

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

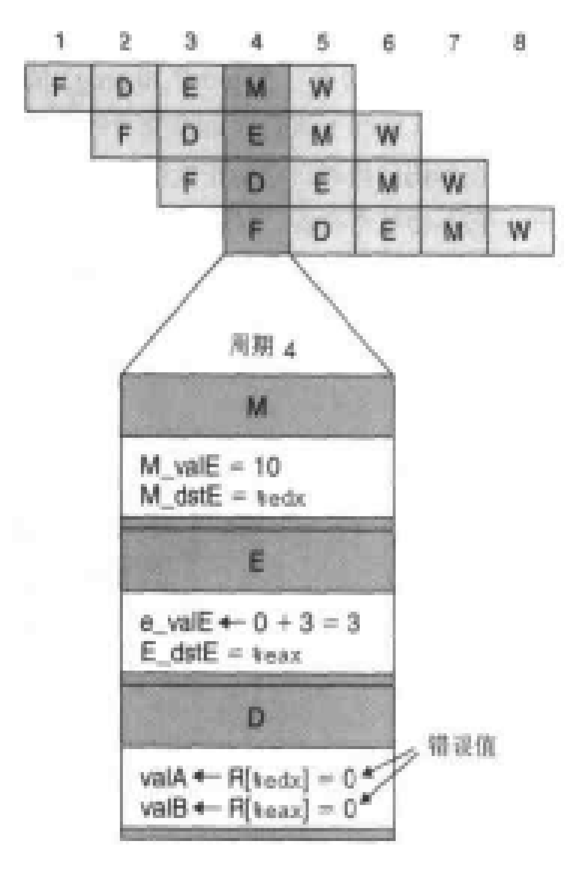


图 4.45 prog4 的流水线化的执行，没有特殊的流水线控制

在周期 4，addl 指令从寄存器文件中读源操作数。对寄存器%edx 的写仍处在访存阶段，而执行阶段正在计算寄存器%eax 的新值。两个操作数 valA 和 valB 得到的都是错误值。

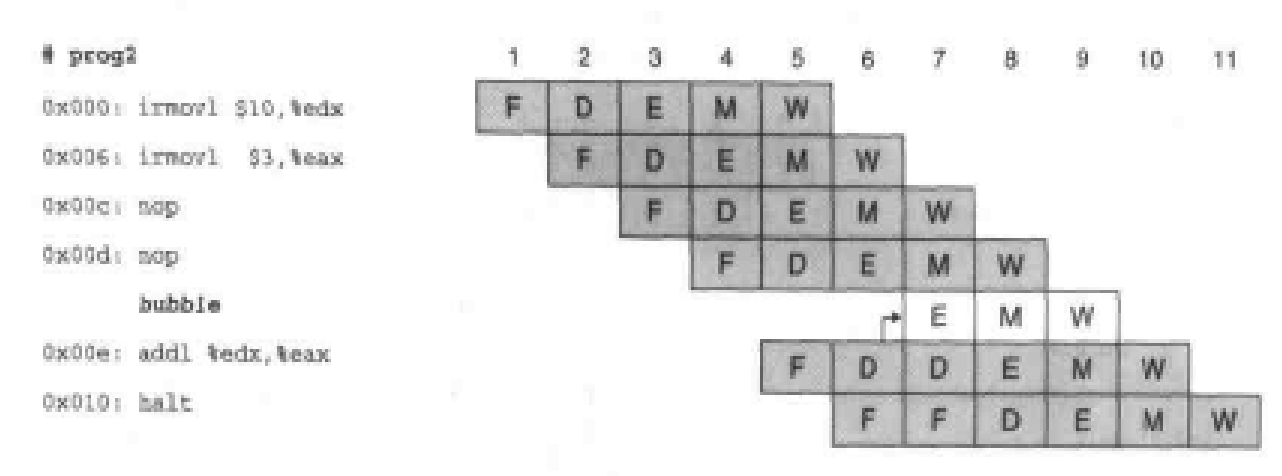


图 4.46 prog2 的使用暂停的流水线化的执行

在周期 6 中对 addl 指令解码之后，暂停控制逻辑发现一个数据冒险，它是由写回阶段中对寄存器%eax 未进行的写造成的。它在执行阶段中插入一个气泡，并在周期 7 中重复对指令 addl 的解码。实际上，机器是动态地插入一条 nop 指令，得到的执行流类似于 prog1 的流（图 4.42）。

将 addl 指令阻塞在解码阶段时，我们还必须将紧跟其后的 halt 指令阻塞在取指阶段。通过将程序计数器保持不变就能做到这一点，这样一来，会不断地对 halt 指令进行取指，直到暂停结束。

暂停技术就是让一组指令阻塞在它们的阶段，而允许其他指令继续通过流水线。在我们的示例

中，我们将 addl 指令阻塞在解码阶段，让 halt 指令阻塞在取指阶段，持续 1~3 个额外的周期，而让两条 irmovl 指令以及 nop 指令（在 prog2 和 prog3 情况中）继续通过执行、访存和写回阶段。那么，我们该让那些在正常情况下该处理 addl 指令的阶段干什么呢？我们解决这个问题的方法是：每次将一条指令阻塞在解码阶段时，都会在执行阶段中插入一个气泡（bubble）。气泡就像一个动态产生的 nop 指令——它不会对寄存器、存储器或条件码产生任何改变。在图 4.46~图 4.48 的流水线图中用白色方框表示。在这些图中，我们用一个 addl 指令的标号为“D”的方框到标号为“E”的方框之间的箭头来表示一个流水线气泡。这些箭头表明，在执行阶段中插入气泡是为了替代 addl 指令，它本来应该经过解码阶段进入执行阶段。在 4.5.9 节中，我们将看到使流水线暂停以及插入气泡的详细机制。

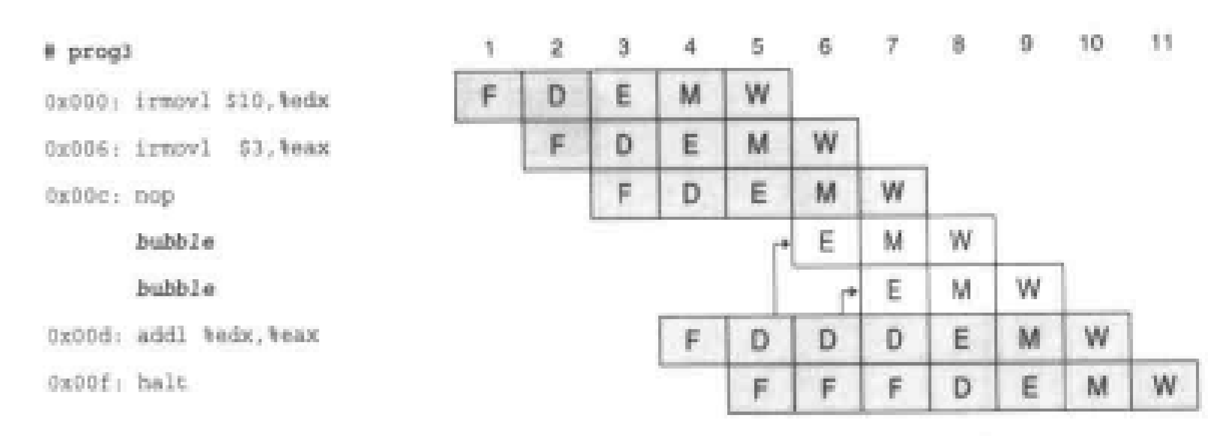


图 4.47 prog3 的使用暂停的流水线化的执行

在周期 5 中对 addl 指令解码之后，暂停控制逻辑发现了对两个源寄存器的数据冒险。它在执行阶段中插入一个气泡，并在周期 6 中重复对指令 addl 的解码。它再次发现对寄存器%eax 的冒险，就在执行阶段中又插入一个气泡，并在周期 7 中重复对指令 addl 的解码。实际上，机器是动态地插入两条 nop 指令，得到的执行流类似于 prog1 的流（图 4.42）。

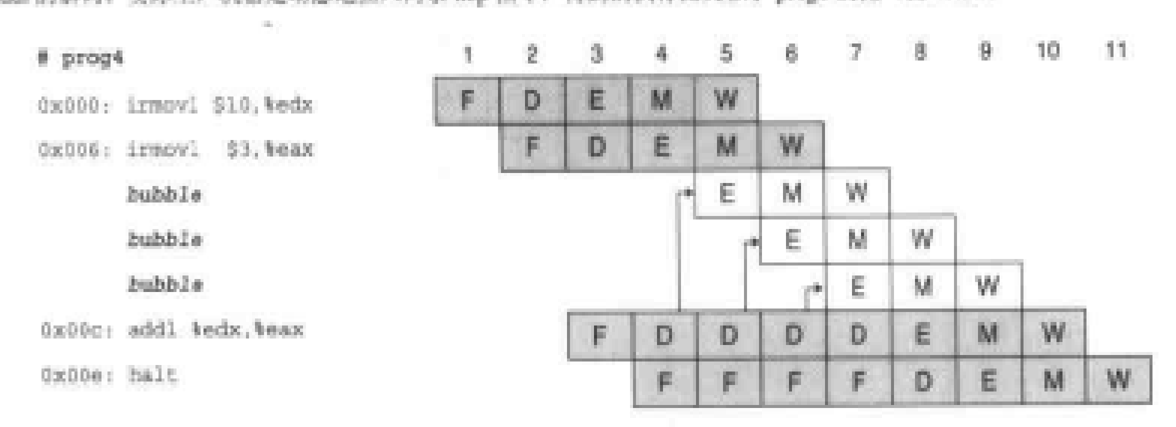


图 4.48 prog4 的使用暂停的流水线化的执行

在周期 4 中对 addl 指令解码之后，暂停控制逻辑发现了对两个源寄存器的数据冒险。它在执行阶段中插入一个气泡，并在周期 5 中重复对指令 addl 的解码。它再次发现对两个源寄存器的冒险，就在执行阶段中插入一个气泡，并在周期 6 中重复对指令 addl 的解码。它再次发现对寄存器%eax 的冒险，就在执行阶段中插入一个气泡，并在周期 7 中重复对指令 addl 的解码。实际上，机器是动态地插入三条 nop 指令，得到的执行流类似于 prog1 的流（图 4.42）。

在使用暂停技术来解决数据冒险中，我们通过动态地产生和 prog1 流（图 4.42）一样的流水线流，有效地执行了程序 prog2、prog3 和 prog4。为 prog2 插入一个气泡，为 prog3 插入两个气泡，为

prog4 插入三个气泡，与在第二条 irmovl 指令和 addl 指令之间有三条 nop 指令，有相同的效果。虽然实现这一机制相当容易（参考家庭作业 4.36），但是得到的性能并不很好。一条指令更新一个寄存器，紧跟其后的指令恰恰使用被更新的寄存器，像这样的情况不胜枚举。这会导致流水线暂停长达三个周期，严重降低了整个的吞吐量。

4.5.6 用转发（forwarding）来避免数据冒险

我们 PIPE-的设计是在解码阶段从寄存器文件中读入源操作数，但是有可能对这些源寄存器的写要在写回阶段才能进行。与其暂停直到写完成，不如简单地将要写的值传到流水线寄存器 E 作为源操作数。图 4.49 用 prog2 周期 6 的流水线图的详细描述来说明了这一策略。解码阶段逻辑发现，寄存器%eax 是操作数 valB 的源寄存器，而在写端口 E 上还有一个对%eax 的未进行的写。它只要简单地将提供到端口 E 的数据字（信号 W_valE）作为操作数 valB 的值，就能避免暂停。这种将结果值直接从一个流水线阶段传到较早阶段的技术称为*数据转发*（data forwarding，或简称转发）。它使得 prog2 的指令能通过流水线而不需要任何暂停。

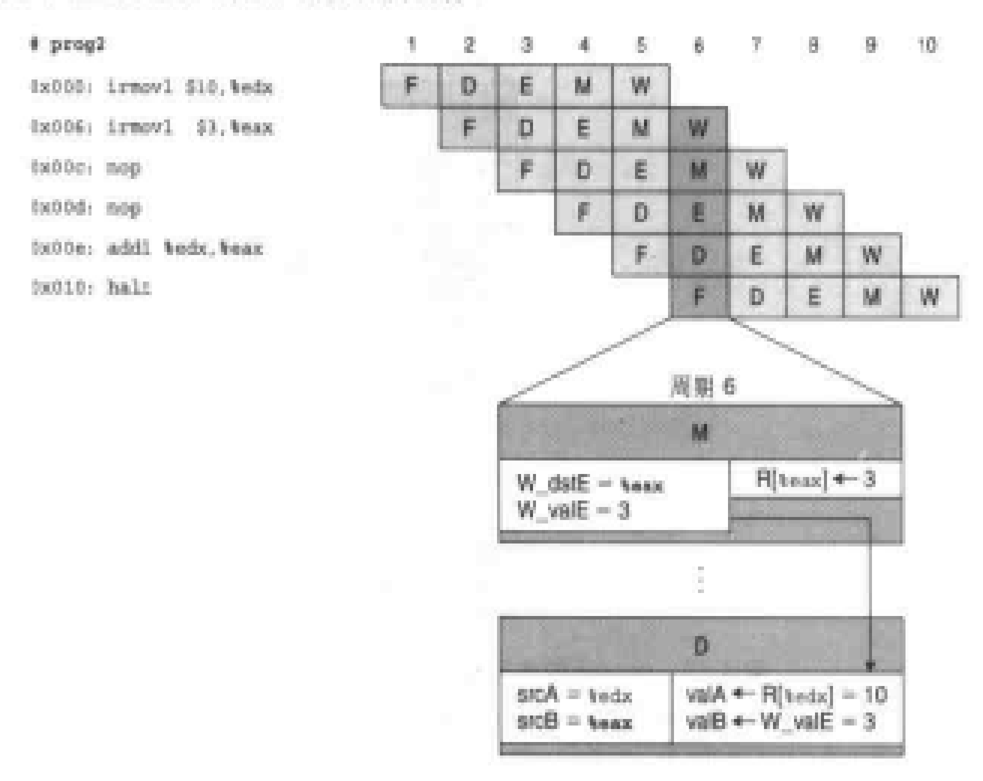


图 4.49 prog2 的使用转发的流水线化的执行

在周期 6 中，解码阶段逻辑发现有在写回阶段中对寄存器%eax 未进行的写。它用这个值，而不是从寄存器文件中读出的值，作为源操作数 valB。

如图 4.50 描述的那样，当访存阶段中有对寄存器未进行的写时，也可以使用数据转发，以避免程序 prog3 中的暂停。在周期 5 中，解码阶段逻辑发现，在写回阶段中端口 E 上有对寄存器%edx 未进行的写，以及在访存阶段中有会在端口 E 上对寄存器%eax 未进行的写。它不会暂停直到这些写真正发生，而是用写回阶段中的值（信号 W_valE）作为操作数 valA，用访存阶段中的值（信号 M_valE）作为操作数 valB。

为了充分利用数据转发技术，我们还可以将新计算出来的值从执行阶段传到解码阶段，以避免

程序prog4所需要的暂停，如图4.51所示。在周期4中，解码阶段逻辑发现在访存阶段中有对寄存器%edx未进行的写，而且执行阶段中ALU正在计算的值稍后也会写入寄存器%eax。它可以将访存阶段中的值（信号M_valE）作为操作数valA，也可以将ALU的输出（信号e_valE）作为操作数valB。注意，使用ALU的输出不会造成任何同步问题。解码阶段只要在时钟周期结束之前产生信号valA和valB，这样在时钟上升开始下一个周期时，流水线寄存器E就能装载来自解码阶段的值了。而在此之前ALU的输出已经是合法的了。

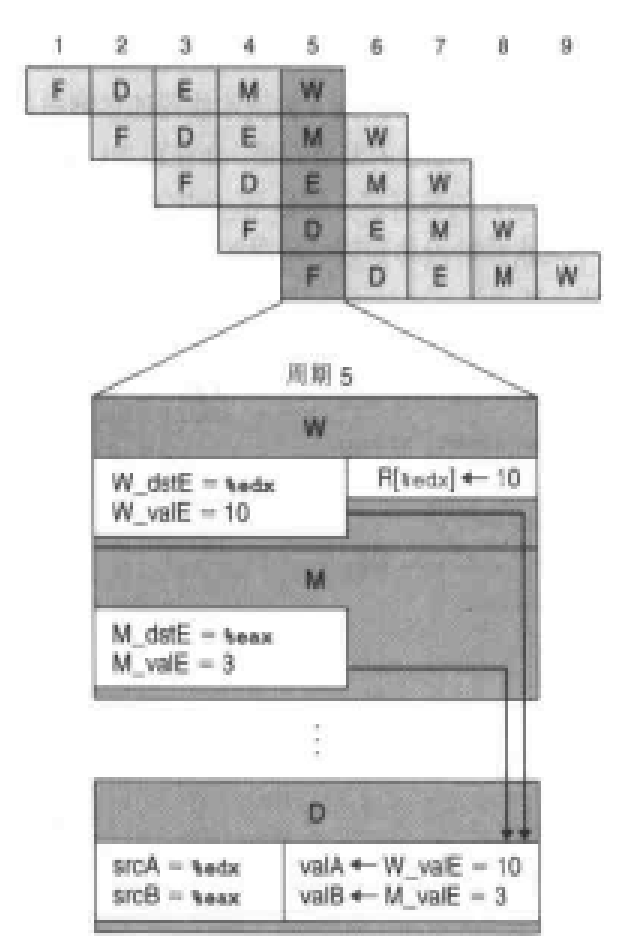


图4.50　prog3的使用转发的流水线化的执行

在周期5中，解码阶段逻辑发现有在写回阶段中对寄存器%edx未进行的写，以及在访存阶段中对寄存器%eax未进行的写。它用这些值，而不是从寄存器文件中读出的值，作为valA和valB的值。

程序prog2～prog4中描述的转发技术的使用都是将ALU产生的以及其目标为写端口E的值进行转发，其实也可以转发从存储器中读出的以及其目标为写端口M的值。从访存阶段，我们可以转发刚刚从数据存储器中读出的值（信号m_valM）。从写回阶段，我们可以转发对端口M未进行的写（信号W_valM）。这样一共就有五个不同的转发源（e_valE、m_valM、M_valE、W_valM和W_valE），以及两个不同的转发目的（valA和valB）。

图4.49～图4.51的扩展图还表明解码阶段如何确定是要用来自寄存器文件的值，还是要用转发过来的值。与每个要写回寄存器文件的值相关的是目的寄存器ID。逻辑会将这些ID与源寄存器ID srcA和srcB相比较，以此来发现是否需要转发。可能有多个目的寄存器ID与一个源ID相等。要解决这样的问题，我们必须在各个转发源中建立起优先级关系。在学习转发逻辑的详细设计时，我

们会讨论这个内容。

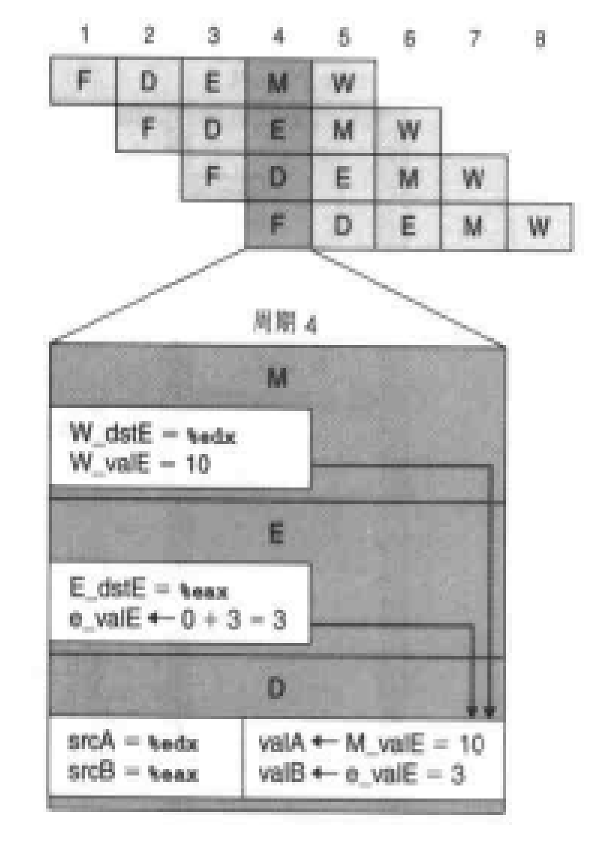


图 4.51　prog4 的使用转发的流水线化的执行

在周期 4 中，解码阶段逻辑发现有在访存阶段中对寄存器%edx 未进行的写，还发现在执行阶段中正在计算寄存器%eax 的新值。它用这些值，而不是从寄存器文件中读出的值，作为 valA 和 valB 的值。

图 4.52 给出的是 PIPE 的抽象结构，它是 PIPE-的扩展，能通过转发处理数据冒险。我们可以看到添加了从五个转发源到解码阶段的额外的反馈路径（用深灰色表示）。这些旁路路径（bypass path）反馈到解码阶段中一个标号为“Forward”的块。这个块会用从寄存器文件中读出的值或转发过来的值作为源操作数 valA 和 valB。

图 4.53 给出了 PIPE 硬件结构的一个更为详细的说明。将这幅图与 PIPE-的结构（图 4.41）相比，我们可以看到来自五个转发源的值反馈到解码阶段中两个标号为“Sel+Fwd A”和“Fwd B”的块。标号为“Sel+Fwd A”的块是 PIPE-中标号为“Select A”的块的功能与转发逻辑的结合。它允许流水线寄存器 M 的 valA 为已增加的程序计数器值 valP，从寄存器文件 A 端口读出的值，或者某个转发过来的值。标号为“Fwd B”的块实现的是源操作数 valB 的转发逻辑。

4.5.7　加载/使用（load/use）数据冒险

有一类数据冒险不能单纯用转发来解决，因为存储器读是在流水线较后面才发生的。图 4.54 举例说明了*加载/使用冒险*（load/use hazard），其中一条指令（位于地址 0x018 的 mrmovl）从存储器中读出寄存器%eax 的值，而下一条指令（位于地址 0x01e 的 addl）需要该值作为源操作数。图的下部是周期 7 和 8 的扩展说明。addl 指令在周期 7 中需要该寄存器的值，但是 mrmovl 指令直到周期 8 才产生出这个值。为了从 mrmovl“转发到”addl，转发逻辑不得不将值送回到过去的时间！这显然是不可能的，我们必须找到其他机制来解决这种形式的数据冒险。注意，位于地址

0x00c 的 irmovl 指令产生的寄存器%ebx 的值，可以从访存阶段转发到周期 7 处于解码阶段中的 addl 指令。

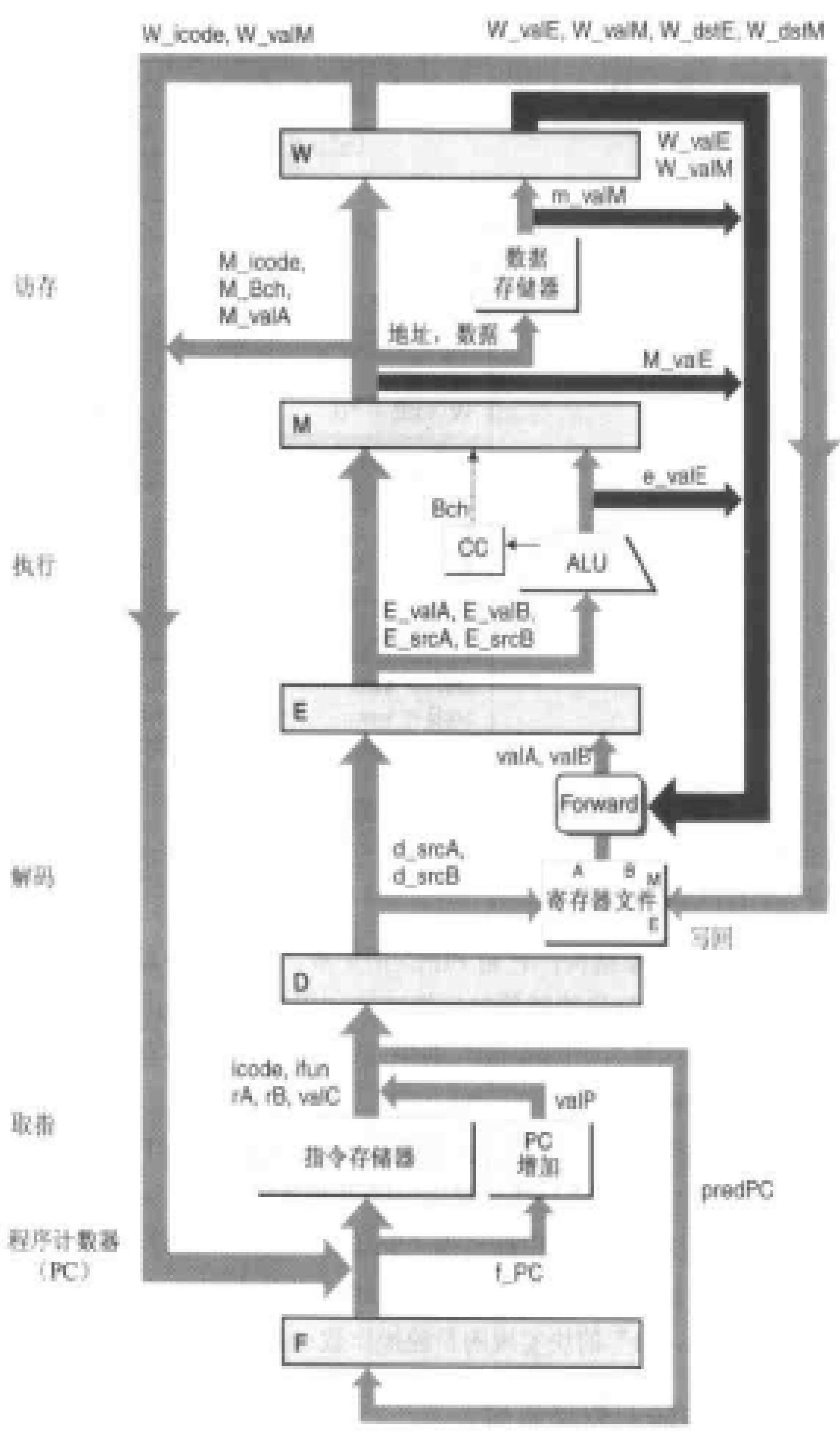


图 4.52 我们最终的流水线化的实现——PIPE 的抽象图

新添加的旁路路径（用深灰色表示）使得可以转发来自前面三条指令的结果。这使得我们可以处理大部分形式的数据冒险，而不需要暂停流水线。

如图 4.55 展示的那样，我们可以通过将暂停和转发结合起来，避免加载/使用数据冒险。当 mrmovl 指令通过执行阶段时，流水线控制逻辑发现解码阶段中的指令（addl）需要这个从存储器中

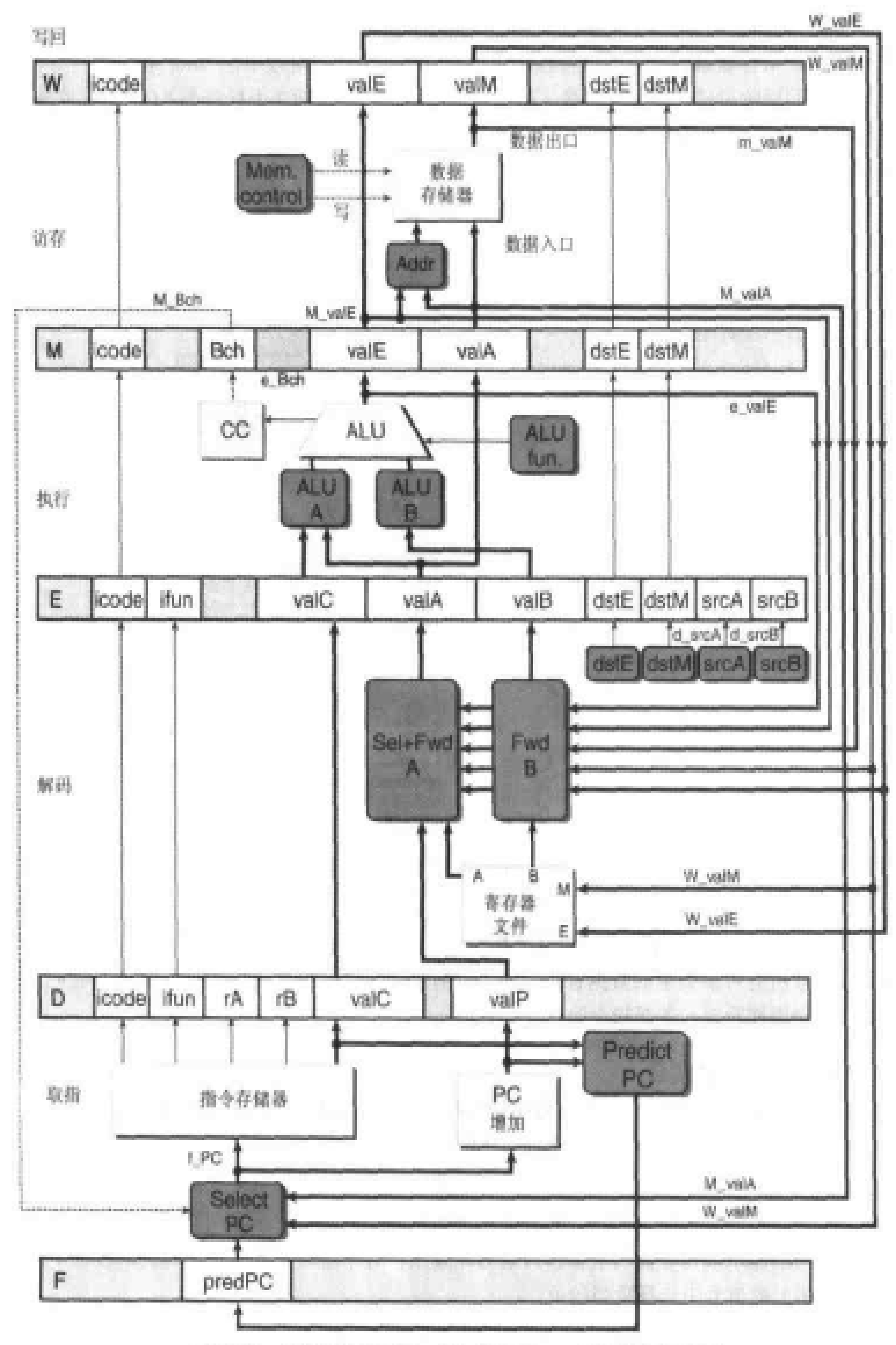


图 4.53 我们最终的流水线化的实现——PIPE 的硬件结构

有些连接没有画出来。

读出的结果。它会将解码阶段中的指令暂停一个周期，导致执行阶段中插入一个气泡。如周期 8 的扩展说明所示，从存储器中读出的值可以从访存阶段转发到解码阶段中的 addl 指令。寄存器%edx 的值也可以从写回阶段转发到访存阶段。就像流水线图中，从周期 7 中标号为“D”的方框到周期 8 中标号为“E”的方框的箭头表明的那样，插入的气泡代替了正常情况下本来应该继续通过流水线的 addl 指令。

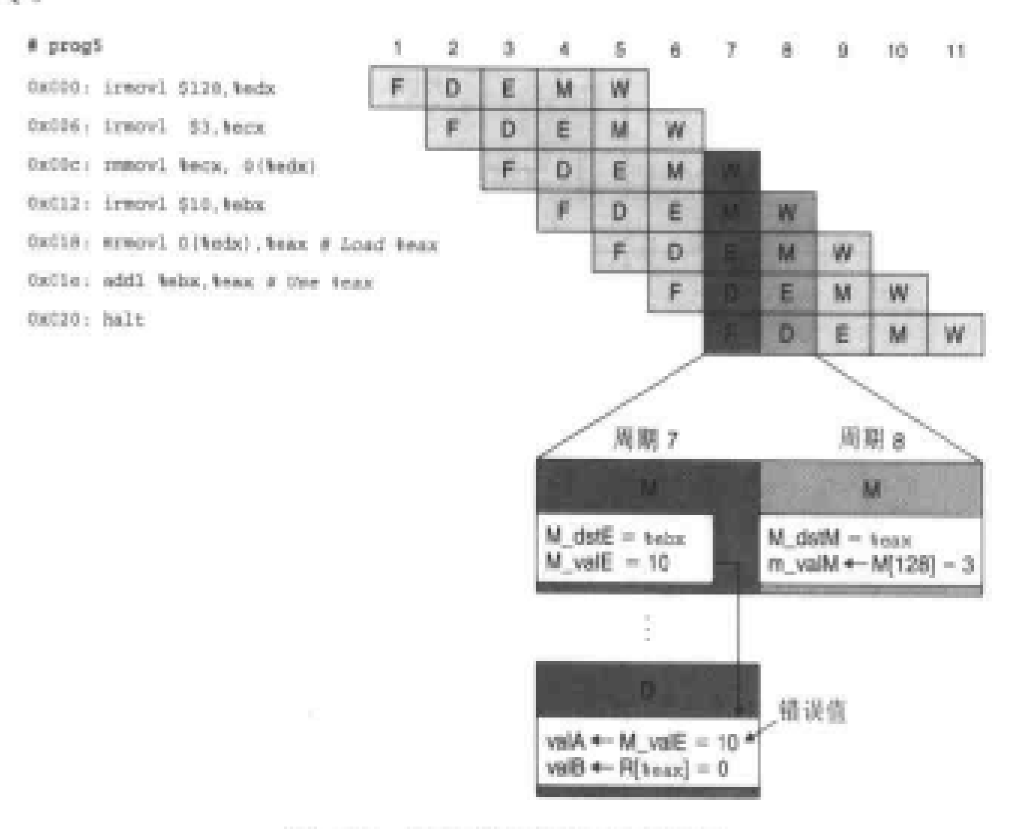


图 4.54 加载/使用数据冒险的示例

addl 指令在周期 7 解码阶段中需要寄存器%eax 的值。前面的 mrmovl 指令在周期 8 访存阶段中读出寄存器%eax 的新值，这对于 addl 指令来说太迟了。

这种用暂停来处理加载/使用冒险的方法称为*加载互锁*（load interlock）。加载互锁和转发技术结合起来足以处理所有可能类型的数据冒险。因为只有加载互锁会降低流水线的吞吐量，我们几乎可以实现每个时钟周期发射一条新指令的吞吐量目标。

4.5.8 PIPE 各阶段的实现

现在我们已经创建了 PIPE（即我们使用转发技术的流水线化的 Y86 处理器）的整体结构。它使用了一组与前面顺序设计相同的硬件单元，另外增加了一些流水线寄存器、重新配置了的逻辑块，以及流水线控制逻辑。在本节中，我们将浏览各个逻辑块的设计，而将流水线控制逻辑的设计放到下一节中再讲。许多逻辑块与 SEQ 和 SEQ+中相应部件完全相同，除了我们必须从来自不同流水线寄存器（用大写的流水线寄存器的名字作为前缀）或来自各个阶段计算（用小写的阶段名字的第一个字母作为前缀）的信号中选择适当的值。

作为一个示例，比较一下 SEQ 中产生 srcA 信号的逻辑的 HCL 代码与 PIPE 中相应的代码：

```
# Code from SEQ
```

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register

];
# Code from PIPE
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

图 4.55 用暂停来处理加载/使用冒险

通过将 addl 指令在解码阶段暂停一个周期，就可以将 valB 的值从访存阶段中的 mrmovl 指令转发到解码阶段中的 addl 指令。

它们的不同之处只在于 PIPE 信号都加上了前缀“D_”，以表明信号是来自流水线寄存器 D。为了避免重复，我们在此就不列出那些与 SEQ 中代码只有名字前缀不同的块的 HCL 代码。不过作为参考，附录 A 的 A.4 节中列出了 PIPE 的完整 HCL 代码。

PC 选择和取指阶段

图 4.56 提供了 PIPE 取指阶段逻辑的一个详细描述。像前面讨论过的那样，这个阶段还必须选

择程序计数器的当前值，以及预测下一个 PC 值。用于从存储器中读取指令和抽取不同指令字段的硬件单元与 SEQ 中考虑的那些一样（参见 4.3.4 节中的取指阶段）。

PC 选择逻辑从三个程序计数器源中进行选择。当一条预测错误的分支进入访存阶段时，会从流水线寄存器 M（信号 M_valA）中读出该指令 valP 的值（指明下一条指令的地址）。当 ret 指令进入写回阶段时，会从流水线寄存器 W（信号 W_valM）中读出返回地址。其他情况会使用存放在流水线寄存器 F 中（信号 F_predPC）的 PC 的预测值。

```
int f_pc = [
        # Mispredicted branch. Fetch at incremented PC
        M_icode == IJXX && !M_Bch : M_valA;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

图 4.56 PIPE 的 PC 选择和取指逻辑

在一个周期的时间限制内，处理器只能预测下一条指令的地址。

当取出的指令为函数调用或跳转时，PC 预测逻辑会选择 valC，否则就会选择 valP：

```
int new_F_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
```

```
        1 : f_valP;
];
```

标号为“Instr valid”、“Need regids”和“Need valC”的逻辑块和SEQ中的一样，同时对信号做适当的重命名。

解码和写回阶段

图4.57给出的是PIPE的解码和写回逻辑的详细说明。标号为“dstE”、“dstM”、“srcA”和“srcB”的块非常类似于它们在SEQ的实现中的相应部件。我们观察到，提供给写端口的寄存器ID来自于写回阶段（信号W_dstE和W_dstM），而不是来自于解码阶段。这是因为我们希望进行写的目的寄存器是由写回阶段中的指令指定的。

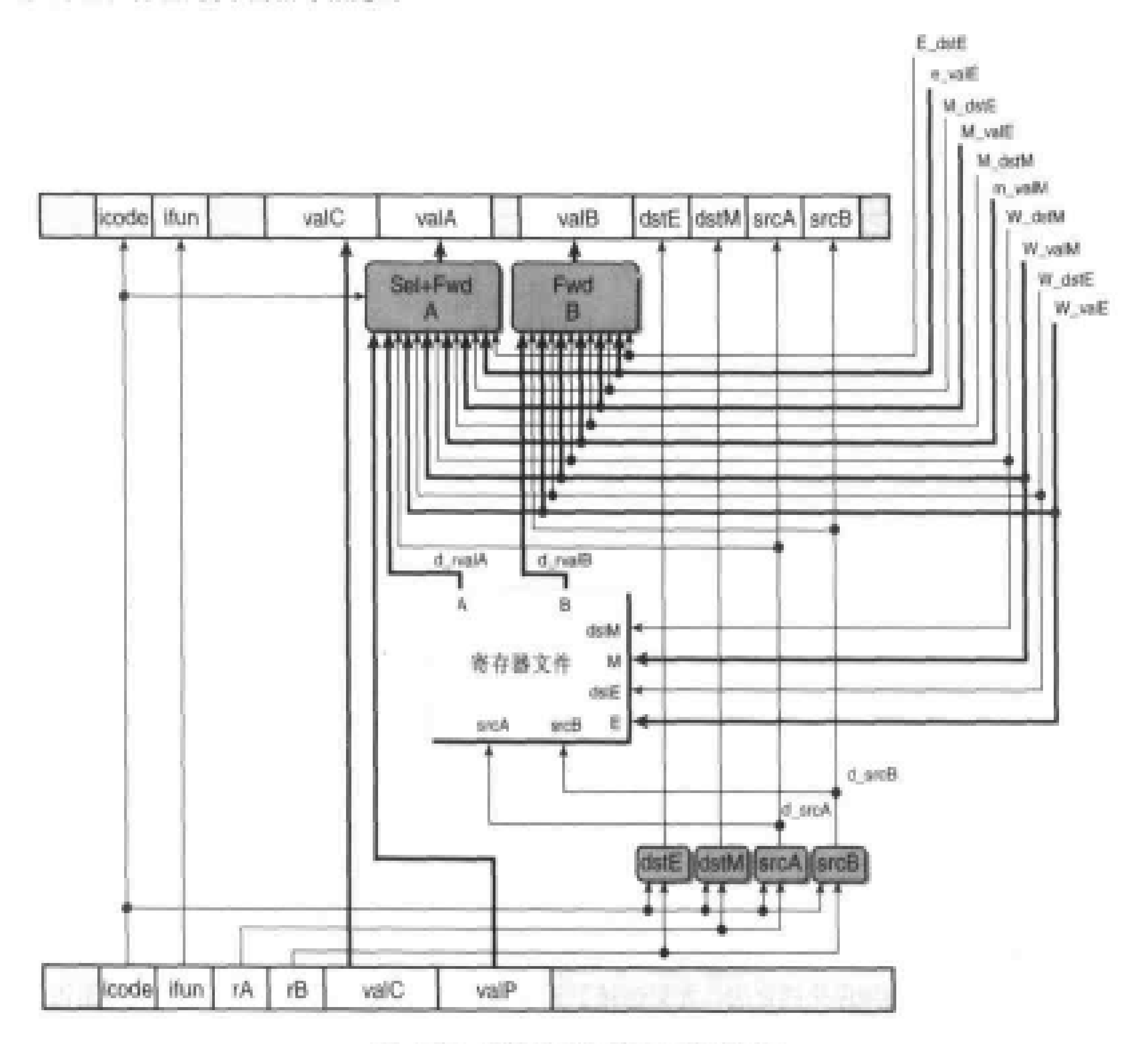


图 4.57 PIPE的解码和写回阶段逻辑

没有指令既需要valP又需要来自寄存器端口A中读出的值，因此对后面的阶段来说，这两者可以合并为信号valA。标号为“Sel+Fwd A”的块执行该任务，并实现源操作数valA的转发逻辑。标号为“Fwd B”的块实现源操作数valB的转发逻辑。寄存器写的位置是由来自写回阶段的dstA和dstB信号指定的，而不是来自于解码阶段，因为它要写的是当前正在写回阶段中的指令的结果。

练习题 4.23

解码阶段中标号为"dstE"的块根据来自流水线寄存器D中取出的指令的各个字段，产生dstE信号。在PIPE的HCL描述中，得到的信号命名为new_E_dstE。根据SEQ信号dstE的HCL描述，写出这个信号的HCL代码。（参考4.3.4节中的解码阶段。）

这个阶段的复杂性主要是跟转发逻辑相关。就像前面提到的那样，标号为"Sel+Fwd A"的块扮演两个角色。它为后面的阶段将valP信号合并到valA信号，这样可以减少流水线寄存器中的状态数量。它还实现了源操作数valA的转发逻辑。

合并信号valA和valP应用了这样一个事实，那就是只有call和跳转指令在后面的阶段中需要valP的值，而这些指令并不需要从寄存器文件A端口中读出的值。这个选择是由该阶段的icode信号来控制的。当信号D_icode与call或jXX的指令代码相匹配时，这个块就会选择D_valP作为它的输出。

如4.5.6节中提到的那样，有五个不同的转发源，每个都有一个数据字和一个目的寄存器ID：

数据字	寄存器 ID	源描述
e_valE	E_dstE	ALU 输出
m_valM	M_dstM	存储器输出
M_valE	M_dstE	访存阶段中对端口 E 未进行的写
W_valM	W_dstM	写回阶段中对端口 M 未进行的写
W_valE	W_dstE	写回阶段中对端口 E 未进行的写

如果不满足任何转发条件，这个块就会选择d_rvalA（即从寄存器端口A中读出的值）作为它的输出。

综上所述，我们得到下面这样的流水线寄存器E的valA的新值的HCL描述：

```
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == E_dstE : e_valE;   # Forward valE from execute
        d_srcA == M_dstM : m_valM;   # Forward valM from memory
        d_srcA == M_dstE : M_valE;   # Forward valE from memory
        d_srcA == W_dstM : W_valM;   # Forward valM from write back
        d_srcA == W_dstE : W_valE;   # Forward valE from write back
        1 : d_rvalA;   # Use value read from register file
];
```

上述HCL代码中赋予这五个转发源的优先级是非常重要的。这种优先级是由HCL代码中检测五个目的寄存器ID的顺序来确定的。如果选择了其他顺序，对某些程序来说，流水线就会出错。图4.58给出了一个程序示例，它要求对执行和访存阶段中的转发源设置正确的优先级。在这个程序中，前两条指令写寄存器%edx，而第三条指令用这个寄存器作为它的源操作数。当指令rrmovl在周期4到达解码阶段时，转发逻辑必须在两个都以该源寄存器为目的的值中选择一个。它应该选择哪一个呢？为了设定优先级，我们必须考虑当一次执行一条指令时，机器语言程序的行为。第一条irmovl指令会将寄存器%edx设为10，第二条irmovl指令会将之设为3，然后rrmovl指令会从%edx

中读出 3。为了模拟这种行为，我们的流水线化的实现应该总是给处于最早流水线阶段中的转发源以较高的优先级，因为它保持着程序序列中设置该寄存器的最近的指令。因此，上述 HCL 代码中的逻辑首先会检测执行阶段中的转发源，然后是访存阶段中的，最后才是写回阶段中的。

对同在访存或写回阶段中的两个源之间的转发优先级，只对指令 popl %esp 有影响，因为只有这条指令能同时写两个寄存器。

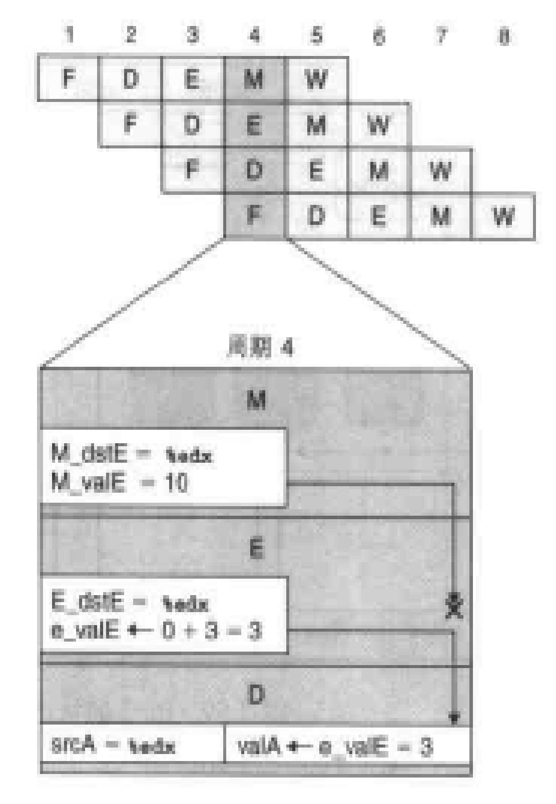


图 4.58 转发优先级的说明

在周期 4 中，%edx 的值既可以从执行阶段也可以从访存阶段中得到。转发逻辑应该选择执行阶段中的值，因为它代表最近产生的该寄存器的值。

练习题 4.24

假设 new_E_valA 的 HCL 代码中第三和第四种情况（来自访存阶段的两个转发源）的顺序是反过来的。请描述下列程序中 rrmovl 指令（第 5 行）造成的行为：

```
1 irmovl $5, %edx
2 irmovl $0x100,%esp
3 rmmovl %edx,0(%esp)
4 popl %esp
5 rrmovl %esp,%eax
```

练习题 4.25

假设 new_E_valA 的 HCL 代码中第五和第六种情况（来自写回阶段的两个转发源）的顺序是反过来的。写出一个会运行错误的 Y86 程序。请描述错误是如何发生的，以及它对程序行为的影响。

练习题 4.26

根据提供到流水线寄存器 E 的源操作数 valB 的值，写出信号 new_E_valB 的 HCL 代码。

执行阶段

图 4.59 展现的是 PIPE 执行阶段的逻辑。这些硬件单元和逻辑块同 SEQ 中的相同，同时对信号做适当的重命名。我们可以看到信号 e_valE 和 E_dstE 作为转发源，指向解码阶段。

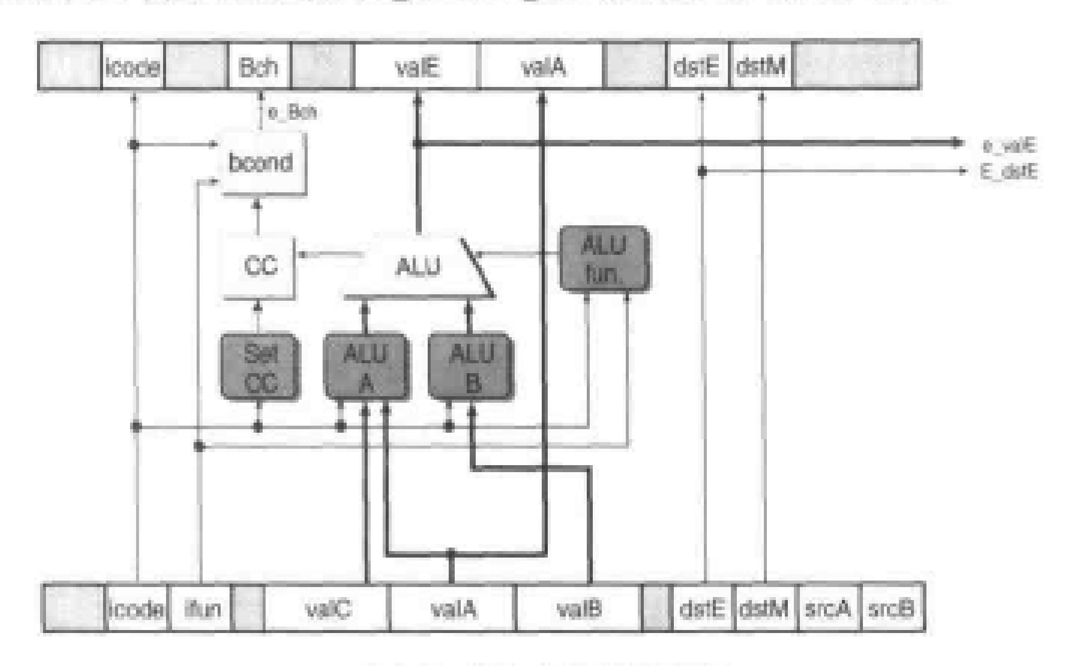


图 4.59 PIPE 的执行阶段逻辑

这一部分的设计与 SEQ 实现中的逻辑设计非常相似。

访存阶段

图 4.60 给出的是 PIPE 的访存阶段逻辑。将这个逻辑与 SEQ 的访存阶段（图 4.28）相比较，我们看到，正如前面提到的那样，PIPE 中没有 SEQ 中标号为"Data"的块。这个块是用来在数据源 valP（对 call 指令来说）和 valA 中进行选择的，但是这个选择现在是由解码阶段中标号为"Sel+Fwd A"的块来执行的。这个阶段中的其他块都和 SEQ 中相应的部件相同，同时对信号做适当的重命名。在这张图中，你还可以看到许多流水线寄存器中的值，同时 M 和 W 还作为转发和流水线控制逻辑的一部分，提供给电路中其他部分。

4.5.9 流水线控制逻辑

现在我们准备要创建流水线控制逻辑，完成我们的 PIPE 设计了。这个逻辑必须处理下面三种控制情况，即其他机制（例如数据转发和分支预测）不足以处理的控制情况：

处理 ret：流水线必须暂停直到 ret 指令到达写回阶段。

加载/使用冒险：在一条从存储器中读出一个值的指令和一条使用该值的指令之间，流水线必须暂停一个周期。

预测错误的分支：在分支逻辑发现不应该选择分支之前，分支目标处的几条指令已经进入流水线了。必须从流水线中去掉这些指令。

我们会浏览每种情况所期望的行为，然后再设计出处理这些情况的控制逻辑。

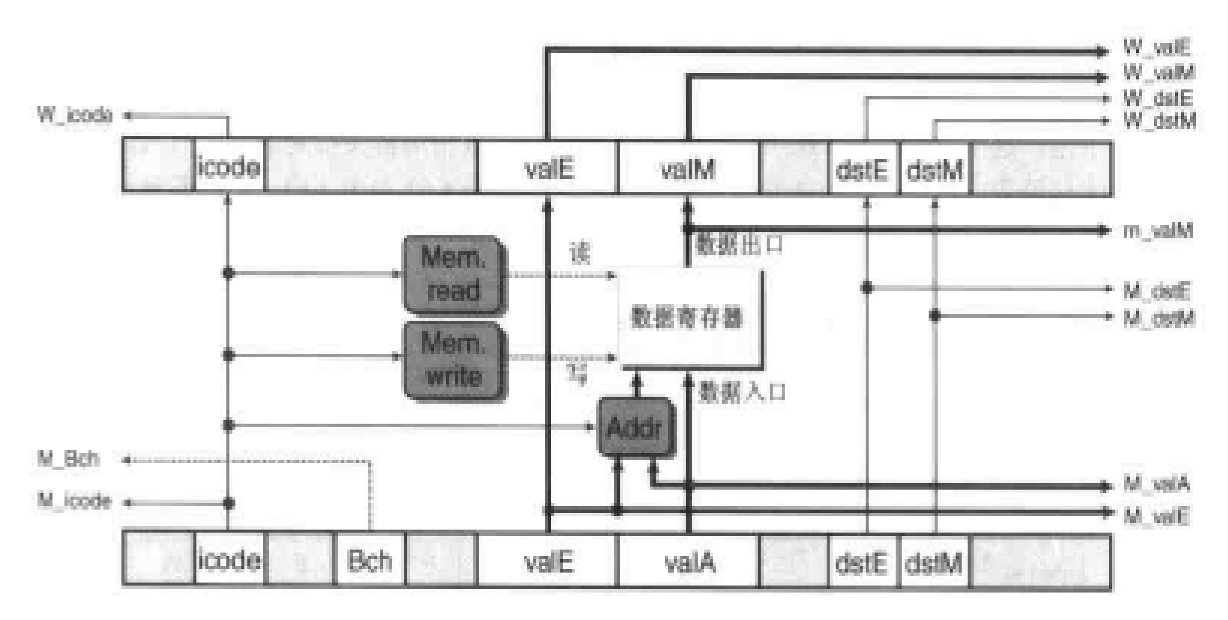


图 4.60 PIPE 的访存阶段逻辑

许多从流水线寄存器 M 和 W 来的信号被传递到较早的阶段，以提供写回的结果、指令地址以及转发的结果。

特殊控制情况所期望的处理

对于 ret 指令，考虑下面的示例程序。这个程序是用汇编代码表示的，左边是各个指令的地址，以供参考：

```
0x000:    irmovl Stack,%esp       # Intialize stack pointer
0x006:    call proc               # Procedure call
0x00b:    irmovl $10,%edx         # Return point
0x011:    halt
0x020: .pos 0x20
0x020:    proc:
0x020:    ret                     # proc:
0x021:    rrmovl %edx,%ebx        # Not executed
0x030: .pos 0x30
0x030: Stack:                     # Stack: Stack pointer
```

图 4.61 给出了我们希望流水线如何来处理 ret 指令。同前面的流水线图一样，这幅图展示了流水线的活动，时间是从左向右增加的。与前面不同的是，指令列出的顺序与它们在程序中出现的顺序并不相同，这是因为这个程序含有一个控制流，指令并不是按线性顺序执行的。看看指令的地址就能看出它们在程序中的位置。

如图 4.61 所示，周期 3 中取出 ret 指令，并沿着流水线前进，在周期 7 进入写回阶段。在它经过解码、执行和访存阶段时，流水线不能做任何有用的活动。我们只能在流水线中插入三个气泡。一旦 ret 指令到达写回阶段，PC 选择逻辑就会将程序计数器设为返回地址，然后取指阶段就会取出位于返回点（地址 0x00b）处的 irmovl 指令。

图 4.62 给出的是示例程序中 ret 指令的实际处理过程。在此我们可以看到，没有办法在流水线的取指阶段中插入气泡。每个周期，取指阶段从指令存储器中读出一条指令。看看 4.5.8 节中实现 PC 预测逻辑的 HCL 代码，我们可以看到，对 ret 指令来说，PC 的新值被预测成 valP 的，也就是下一条指令的地址。在我们的示例程序中，会是 0x021，即 ret 后面 rrmovl 指令的地址。对这个例子

来说，这种预测是不对的，即使对大部分情况来说，也是不对的，但是在我们的设计中，我们并不试图正确预测返回地址。取指阶段会暂停三个时钟周期，导致取出 rrmovl 指令，但是在解码阶段就被替换成了气泡。这个过程在图 4.62 中是这样表示的，三个取指用箭头指向下面的气泡，气泡会经过剩下的流水线阶段。最后，周期 7 取出 irmovl 指令。比较图 4.62 和图 4.61，可以看到，我们的实现达到了期望的效果，只不过连续三个周期取出了不正确的指令。

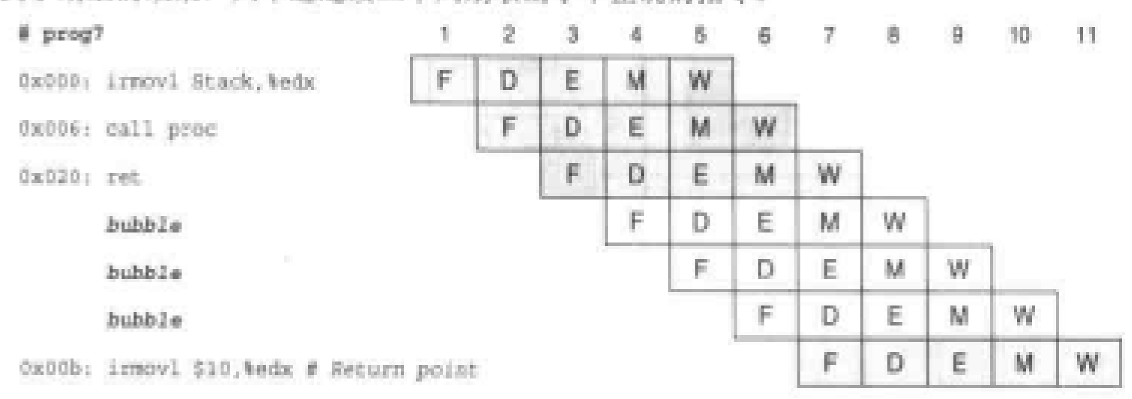


图 4.61 ret 指令处理的简化视图

当 ret 经过解码、执行和访存阶段时，流水线应该暂停，在处理过程中插入三个气泡。一旦 ret 指令到达写回阶段（周期 7），PC 选择逻辑就会选择返回地址作为取指地址。

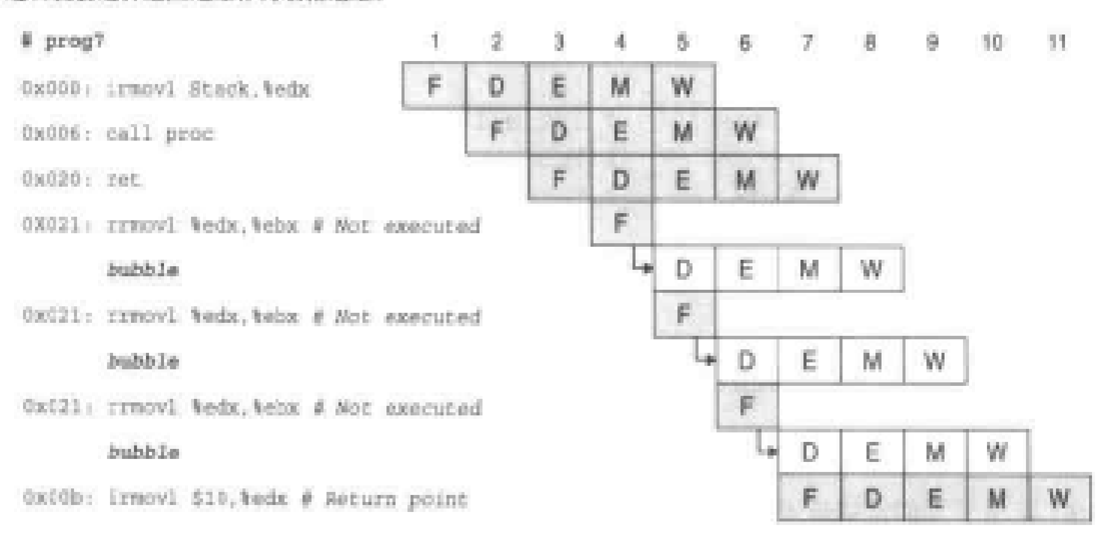


图 4.62 ret 指令处理的实际处理过程

取指阶段反复取出 ret 指令后面的 rrmovl 指令，但是流水线控制逻辑在解码阶段中插入气泡，而不是让 rrmovl 指令继续下去。由此得到的行为与图 4.61 所示的等价。

在 4.5.7 节中，我们已经描述了加载/使用冒险（load/use hazard）所期望的流水线操作，如图 4.55 所示，只有 mrmovl 和 popl 指令会从存储器中读数据。当这两条指令中的一条处于执行阶段，而需要该目的寄存器的指令正处在解码阶段时，我们要将第二条指令阻塞在解码阶段，并在下一个周期往执行阶段中插入一个气泡。此后，转发逻辑会解决这个数据冒险。可以通过将流水线寄存器 D 保持为固定状态，从而将一个指令阻塞在解码阶段。与此同时，还必须将流水线寄存器 F 保持为固定状态，这样一来，就会第二次取出下一条指令。总之，实现这个流水线流需要发现冒险条件（hazard condition），保持流水线寄存器 F 和 D 固定不变，并且在执行阶段中插入气泡。

为了处理预测错误的分支，让我们来考虑下面这个用汇编代码表示的程序，左边是各个指令的地址，以供参考：

```
0x000:    xorl %eax,%eax
0x002:    jne target              # Not taken
0x007:    irmovl $1, %eax         # Fall through
0x00d:    halt
0x00e: target:
0x00e:    irmovl $2, %edx         # Target
0x014:    irmovl $3, %ebx         # Target+1
0x01a: halt
```

图 4.63 表明是如何处理这些指令的。同前面一样，指令是按照它们进入流水线的顺序列出的，而不是按照它们出现在程序中的顺序。因为预测跳转指令会选择分支，所以周期 3 中会取出位于跳转目标处的指令，而周期 4 中会取出该指令后的那条指令。在周期 4，分支逻辑发现不应该选择分支之前，已经取出了两条指令，不应该继续执行下去了。幸运的是，这两条指令都没有导致程序员可见的状态发生改变。只有到指令到达执行阶段时才会发生那种情况，在执行阶段中，指令会改变条件码。我们只要在下一个周期往解码和执行阶段中插入气泡，并同时取出跳转指令后面的指令，这样就能取消——有时也称为指令排除（squashing）——那两条预测错误的指令。这样一来，两条预测错误的指令就会从流水线中消失。

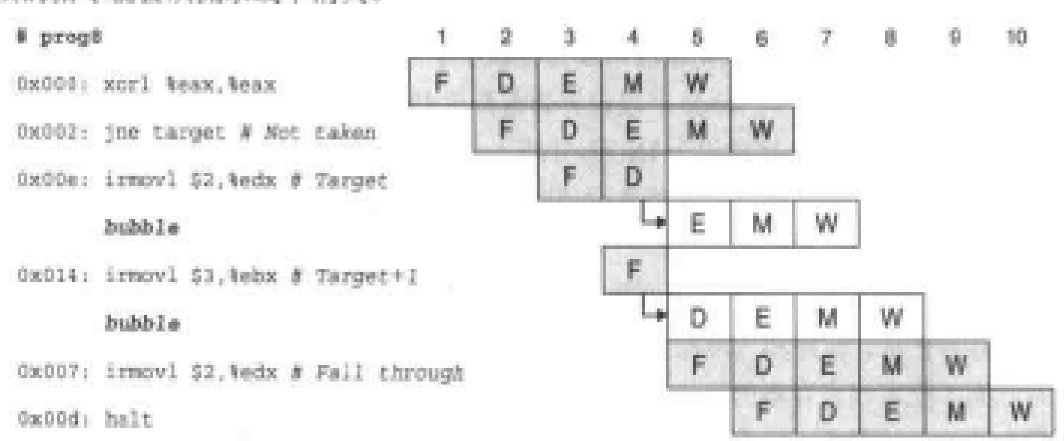


图 4.63 处理预测错误的分支指令

流水线预测会选择分支，所以开始取跳转目标处的指令。在周期 4 发现预测错误之前，已经取出了两条指令，此时，跳转指令正在通过执行阶段。在周期 5 中，流水线往解码和执行阶段中插入气泡，取消了两条目标指令，同时还取出跳转后面的那条指令。

发现特殊控制条件

图 4.64 总结了需要特殊流水线控制的条件，它给出的 HCL 表达式描述了在哪些条件下会出现这三种特殊情况。一些简单的组合逻辑块实现了这些表达式，为了在时钟上升开始下一个周期时控制流水线寄存器的活动，这些块必须在时钟周期结束之前产生出结果。在一个时钟周期内，流水线寄存器 D、E 和 M 分别保持着处于解码、执行和访存阶段中的指令的状态。在到达时钟周期末尾时，信号 d_srcA 和 d_srcB 会被设置为解码阶段中指令的源操作数的寄存器 ID。当 ret 指令通过流水线时，要想发现它，只要检查解码、执行和访存阶段中指令的指令码。发现加载/使用冒险要检查执行阶段中的指令类型（mrmovl 或 popl），并把它的目的寄存器与解码阶段中指令的源寄存器相比较。

当跳转指令在执行阶段时，流水线控制逻辑应该能发现分支预测错误的，这样当指令进入访存阶段时，它就能设置从错误预测中恢复所需要的条件。当跳转指令处于执行阶段时，信号 e_Bch 指明是否要选择分支。

条　件	触　发　器
处理 ret	IRET ∈ {D_icode, E_icode, M_icode}
加载/使用冒险	E_icode ∈ {IMRMOVL,IPOPL}&&E_dstM ∈ {d_srcA,d_srcB}
预测错误的分支	E_icode = IJXX && !e_Bch

图 4.64　流水线控制逻辑的发现条件

三种不同的条件要么会暂停流水线，要么取消部分执行过的指令，从而要改变流水线流。

流水线控制机制

图 4.65 给出的是一些低级机制，它们使得流水线控制逻辑能将指令阻塞在流水线寄存器中，或是往流水线中插入一个气泡。这些机制包括对 4.2.5 节中描述的基本时钟寄存器的小扩展。假设每个流水线寄存器有两个控制输入：暂停（stall）和气泡（bubble）。这些信号的设置决定了当时钟上升时该如何更新流水线寄存器。在正常操作下（情况 A），这两个输入都设为 0，使得寄存器加载它的输入作为新的状态。当暂停信号设为 1 时（情况 B），禁止更新状态。相反，寄存器会保持它以前的状态。这使得它可以将指令阻塞在某个流水线阶段中。当气泡信号设置为 1 时（情况 C），寄存器状态会设置成某个固定的复位配置（reset configuration），得到一个等效于 nop 指令的状态。一个流水线寄存器的复位配置的 0、1 模式是由流水线寄存器中字段的集合决定的。例如，要往流水线寄存器 D 中插入一个气泡，我们要将 icode 字段设置为常数值 INOP（图 4.24）。要往流水线寄存器 E 中插入一个气泡，我们要将 icode 字段设为 INOP，并将 dstE、dstM、srcA 和 srcB 字段设为常数 RNONE。确定复位配置是硬件设计师在设计流水线寄存器时的任务之一，在此我们不会讨论细节。我们会将把气泡和暂停信号都设为 1 看成是出错。

图 4.66 中的表给出了各个流水线寄存器在三种特殊情况下应该采取的行动。对每种情况的处理都是流水线寄存器正常、暂停和气泡操作的某个组合。

在定时方面，流水线寄存器的暂停和气泡控制信号是由组合逻辑块产生的。当时钟上升时，这些值必须是合法的，使得当下一个时钟周期开始时，每个流水线寄存器要么加载，要么暂停，要么产生气泡。有了这个对流水线寄存器设计的小扩展，我们就能用组合逻辑基本构建块、时钟寄存器和随机访问存储器，来实现一个完整的流水线，包括所有的控制。

控制条件的组合

到目前为止，在我们对特殊流水线控制条件的讨论中，我们假设在任意一个时钟周期内，最多只能出现一个特殊情况。在设计系统时，一个常见的毛病是不能处理同时出现多个特殊情况的情形。让我们来分析一下这些可能性。图 4.67 画出了导致特殊控制条件的流水线状态。这些图给出的是解码、执行和访存阶段的块。暗色的方框代表要出现这种条件必须要满足的特别限制。加载/使用冒险要求执行阶段中的指令将一个值从存储器读到寄存器中，同时解码阶段中的值要以该寄存器作为源操作数。预测错误的分支要求执行阶段中的指令是一个跳转指令。对 ret 来说有三种可能的情况——指令可以处在解码、执行或访存阶段。当 ret 指令通过流水线时，前面的流水线阶段都是气泡。

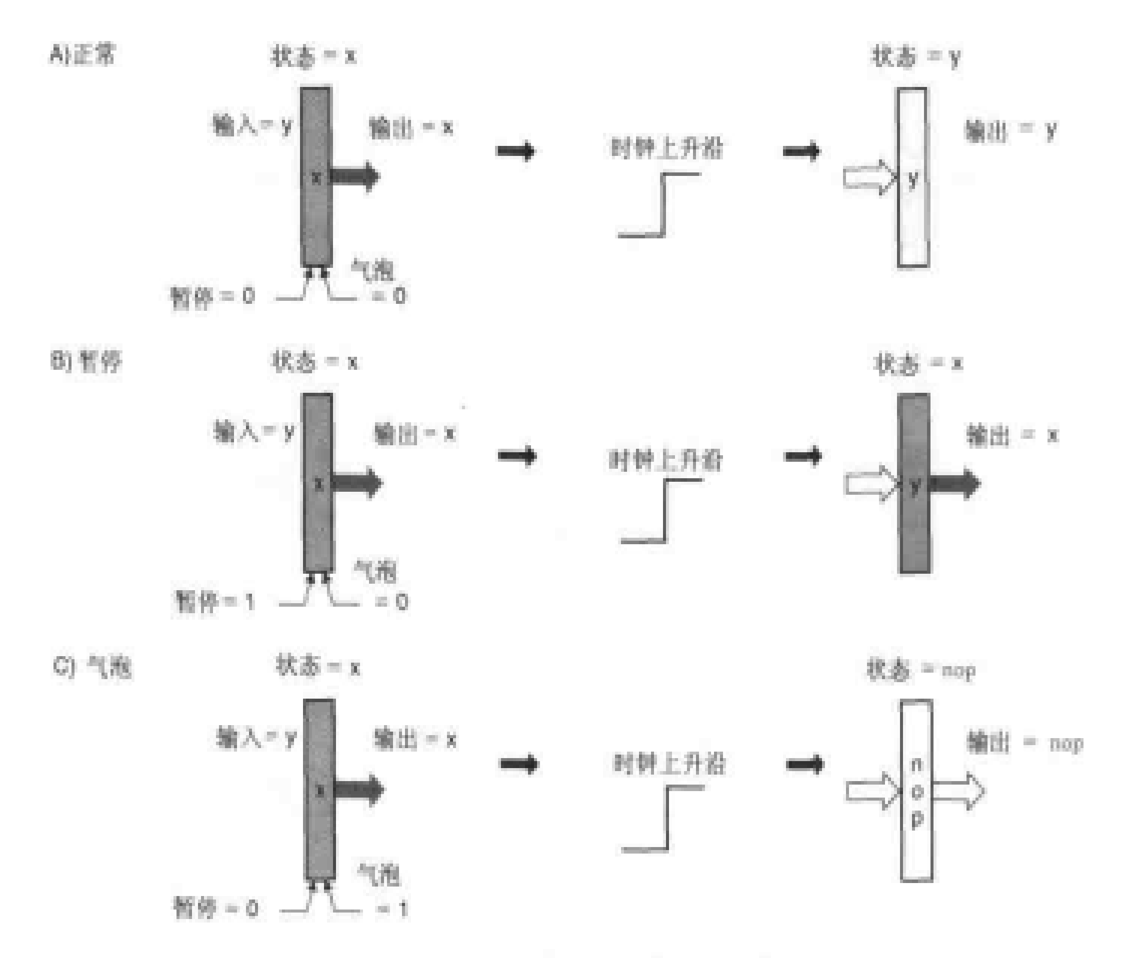


图 4.65 附加的流水线寄存器操作

在正常情况下，A）当时钟上升时，寄存器的状态和输出被设置成输入的值。当运行在暂停模式中时，B）状态保持为先前的值不变。当运行在气泡模式中时，C）会用 nop 操作的状态覆盖当前状态。

	流水线寄存器				
条件	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
预测错误的分支	正常	正常	气泡	正常	正常

图 4.66 流水线控制逻辑的动作

不同的条件需要改变流水线流，或者会暂停流水线，或者会取消部分已执行的指令。

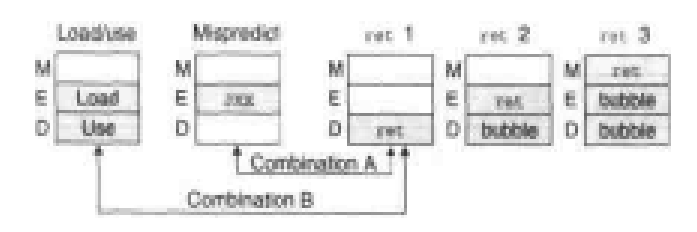


图 4.67 特殊控制条件的流水线状态

图中标明的两对情况可能同时出现。

从这些图中我们可以看出，大多数控制条件是互斥的。例如，不可能同时既有加载/使用冒险又有预测错误的分支，因为一个要求执行阶段中是加载指令（mrmovl 或 popl），而另一个要求其中是一条跳转指令。类似地，第二个和第三个 ret 组合也不可能与加载/使用冒险或预测错误的分支同时出现。只有用箭头标明的两种组合可能同时出现。

组合 A 是指执行阶段中有一条不选择分支的跳转指令，而解码阶段中有一条 ret 指令。出现这种组合要求 ret 位于不选择分支的目标处。流水线控制逻辑应该发现分支预测错误，因此要取消 ret 指令。

练习题 4.27

写一个 Y86 汇编语言程序，它能导致出现组合 A 的情况，并判断控制逻辑是否处理正确。

将对组合 A 条件的控制动作合并起来（图 4.66），我们得到下面这样的流水线控制动作（假设气泡或暂停会覆盖正常的情况）：

	流水线寄存器				
条件	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常
组合	暂停	气泡	气泡	正常	正常

也就是说，组合情况 A 的处理与预测错误的分支相似，只不过在取指阶段是暂停。幸运的是，在下一个周期，PC 选择逻辑会选择跳转后面那条指令的地址，而不是预测的程序计数器值，所以流水线寄存器 F 发生什么是没有关系的。因此我们做出结论，流水线能正确处理这种组合情况。

组合 B 包括一个加载/使用冒险，其中加载指令设置寄存器%esp，然后 ret 指令用这个寄存器作为源操作数，因为它必须从栈中弹出返回地址。流水线控制逻辑应该将 ret 指令阻塞在解码阶段。

练习题 4.28

写一个 Y86 汇编语言程序，它能导致出现组合 B 的情况，并以 halt 指令结束。判断控制逻辑是否处理正确。

	流水线寄存器				
条件	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	暂停	暂停	气泡	正常	正常
组合	暂停	气泡+暂停	气泡	正常	正常
期望的情况	暂停	暂停	气泡	正常	正常

将对组合 B 条件的控制动作结合起来（图 4.66），我们得到下面这样的流水线控制动作：

如果同时触发两组动作，控制逻辑会试图暂停 ret 指令来避免加载/使用冒险，同时又会因为 ret 指令而往解码阶段中插入一个气泡。显然，我们不希望流水线同时执行这两组动作。相反，我们希望它只采取针对加载/使用冒险的动作。处理 ret 指令的动作应该推迟一个周期。

这些分析表明组合 B 需要特殊处理。实际上，我们 PIPE 控制逻辑原来的实现并没有正确处理这种组合情况。即使设计已经通过了许多模拟测试，它还是有细节问题，只有通过刚才那样的分析才能发现出来。当执行一个含有组合 B 的程序时，控制逻辑会将流水线寄存器 D 的气泡和暂停信号都置为 1。这个例子表明了系统分析的重要性。只运行正常的程序是很难发现这个问题的。如果没有发现这个问题，流水线就不能忠实地实现 ISA 的行为。

控制逻辑实现

图 4.68 给出的是流水线控制逻辑的整体结构。根据来自流水线寄存器和流水线阶段的信号，控制逻辑产生流水线寄存器的暂停和气泡控制信号。我们可以将图 4.64 的发现条件和图 4.66 的动作结合起来，产生各个流水线控制信号的 HCL 描述。

遇到加载/使用冒险或 ret 指令，流水线寄存器 F 必须暂停：

```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

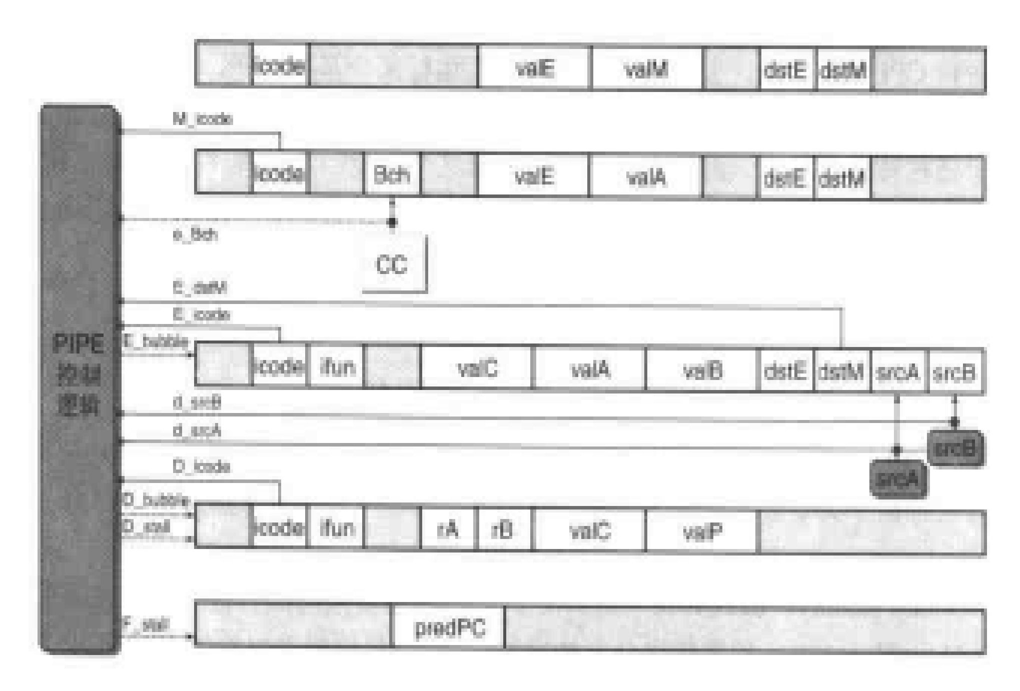


图 4.68 PIPE 流水线控制逻辑

这个逻辑覆盖了通过流水线的正常指令流，以处理特殊条件，例如过程返回、预测错误的分支和加载/使用冒险。

练习题 4.29

写出 PIPE 实现中信号 D_stall 的 HCL 代码。

遇到预测错误的分支或 ret 指令，流水线寄存器 D 必须设置为气泡。不过，正如前面一节中的分析所示，当遇到加载/使用冒险和 ret 指令组合时，不应该插入气泡：

```
bool D_bubble =
```

```
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
   IRET in { D_icode, E_icode, M_icode };
```

练习题 4.30

写出 PIPE 实现中信号 E_bubble 的 HCL 代码。

现在我们就讲完了所有的特殊流水线控制信号的值。在 PIPE 的完整 HCL 代码中，所有其他的流水线控制信号都设为 0。

4.5.10 性能分析

我们可以看到，所有需要流水线控制逻辑进行特殊处理的条件，都会导致我们的流水线不能够实现每个时钟周期发射一条新指令的目标。我们可以通过确定往流水线中插入气泡的频率，来衡量这种效率的损失，因为插入气泡会导致无用的流水线周期。一条返回指令会产生三个气泡，一个加载/使用冒险会产生一个，而一个预测错误的分支会产生两个。我们可以通过计算 PIPE 执行一条指令所需要的平均时钟周期数的估计值，来量化这些处罚对整体性能的影响，这种衡量方法称为 CPI（cycles per instruction，每指令周期数）。这种衡量值是流水线平均吞吐量的倒数，不过时间单位是时钟周期，而不是微微秒。这是对一个设计体系结构效率的很有用的衡量标准。

另一种看待 CPI 的方法是，假设我们在处理器上运行某个基准程序，并观察执行阶段的运行。每个周期，执行阶段要么会处理一条指令，然后这条指令继续通过剩下的阶段，直到完成，要么会处理一个由于三种特殊情况之一而插入的气泡。如果这个阶段一共处理了 C_i 条指令和 C_b 个气泡，那么处理器总共需要大约 C_i+C_b 个时钟周期来执行 C_i 条指令。我们说"大约"是因为我们忽略了启动指令通过流水线的周期。我们可以用如下方法来计算这个基准程序的 CPI：

$$CPI = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

也就是说，CPI 等于 1.0 加上一个处罚项 C_b/C_i，这个项表明执行一条指令平均要插入多少个气泡。因为只有三种指令类型会导致插入气泡，我们可以将这个处罚项分解成三个部分：

$$CPI = 1.0 + lp + mp + rp$$

这里，lp（load penalty，加载处罚）是当由于加载/使用冒险造成暂停时插入气泡的平均数，mp（mispredicted branch penalty，预测错误分支处罚）是当由于预测错误取消指令时插入气泡的平均数，而 rp（return penalty，返回处罚）是当由于 ret 指令造成暂停时插入气泡的平均数。每种处罚都是由该种原因引起的插入气泡的总数（C_b 的一部分）除以执行指令的总数（C_i）。

为了估计每种处罚，我们需要知道相关指令（加载、条件转移和返回）的出现频率，以及对每种指令特殊情况出现的频率。对我们 CPI 的计算，我们使用下面这组频率（等同于[31]和[33]中报告的测量值）：

- 加载指令（mrmovl 和 popl）占所有执行指令的 25%。其中 20%会导致加载/使用冒险。
- 条件分支指令占所有执行指令的 20%。其中 60%会选择分支，而 40%不选择分支。
- 返回指令占所有执行指令的 2%。

因此，我们可以估计每种处罚，它是指令类型频率、条件出现频率和当条件出现时插入气泡数的乘积：

原因	名称	指令频率	条件频率	气泡	乘积
加载/使用	*lp*	0.25	0.20	1	0.05
预测错误	*mp*	0.20	0.40	2	0.16
返回	*rp*	0.02	1.00	3	0.06
总处罚					0.27

三种处罚的总和是 0.27，所以得到 CPI 为 1.27。

我们的目标是设计一个每个周期发射一条指令的流水线，也就是 CPI 为 1.0。我们没有完全达到目标，但是整体性能已经很不错了。我们还能看到，要想进一步降低 CPI，就应该集中注意力在预测错误的分支上。它们占到了整个处罚 0.27 中的 0.16，因为条件转移非常常见，我们的预测策略又经常出错，而每次预测错误都要取消两条指令。

练习题 4.31

假设我们使用了一种成功率可以达到 65%的分支预测策略，例如后向分支选择、前向分支就不选择，如 4.5.3 节中描述的那样。那么对 CPI 有什么样的影响呢？假设其他所有频率都不变。

4.5.11 未完成的工作

我们已经创建了 PIPE 流水线化的微处理器结构，设计了控制逻辑块，并实现了普通流水线流（pipeline flow）不足以处理特殊情况的流水线控制逻辑。不过，PIPE 还是缺乏一些实际微处理器设计中所必需的关键特性。我们会强调其中一些，并讨论要增加这些特性需要些什么。

异常处理

当一个机器级程序遇到出错的情况时，例如，非法指令代码或是指令或数据地址越界会导致程序流中断，称为*异常*（exception）。异常看上去就像过程调用，它调用一个异常处理程序（exception handler），该程序是操作系统的一部分。我们会在第 8 章中详细介绍异常处理。执行 halt 指令也会触发一个异常。异常处理是处理器指令集体系结构的一部分。通常，依赖于异常的类型，异常会导致处理停止。也就是，应该完成到异常点之前的指令，但是该点后的所有指令都不应该对程序员可见的状态产生任何影响。

在一个流水线化的系统中，异常处理包括一些细节问题。首先，可能同时有多条指令会引起异常。例如，在一个流水线操作的周期内，可能会有指令存储器报告取指阶段中指令的指令地址越界、访存阶段中指令的数据地址越界，以及控制逻辑报告解码阶段中指令的非法代码。我们必须确定处理器应该向操作系统报告哪个异常。基本原则是由流水线中最深的指令引起的异常，优先级最高。在上面那个例子中，应该报告访存阶段中指令的地址越界。关于机器语言程序，访存阶段中的指令本来应该在解码或取指阶段中的指令开始之前就结束的，所以，只应该向操作系统报告这个异常。

第二个细节问题是，当首先取出一条指令，开始执行时，导致了一个异常，而后来由于分支预测错误，取消了该指令。下面就是一个这样程序示例的目标代码：

```
0x000: 6300          | xorl %eax,%eax
0x002: 740e000000    | jne Target       # Not taken
```

```
0x007: 308001000000 | irmovl $1, %eax # Fall through
0x00d: 10           | halt
0x00e:              | Target:
0x00e: ff           | .byte 0xFF       # Invalid instruction code
```

在这个程序中，流水线会预测不选择分支，因此它会取出并以一个值为 0xFF 的字节作为指令（由汇编代码中.byte 指示字产生的）。解码阶段会因此发现一个非法指令异常。稍后，流水线会发现不应该选择分支，因此根本就不应该取出位于地址 0x00e 的指令。流水线控制逻辑会取消该指令，但是我们想要避免出现异常。

第三个细节问题的产生是因为流水线化的处理器会在不同的阶段更新系统状态的不同部分。有可能会出现这样的情况，一条指令导致了一个异常，它后面的指令在产生异常的指令完成之前改变了部分状态。比如说，考虑下面的代码序列，其中我们假设不允许用户程序访问大于 0xc0000000 的地址（跟第 10 章中讨论的现在 Linux 中的情况一样）：

```
1    irmovl $0,%esp       # Set stack pointer to 0
2    pushl %eax           # Attempts to write to 0xfffffffc
3    addl %ecx,%eax       # Sets condition codes
```

pushl 指令导致一个地址异常，因为减小栈指针会导致它绕回（wrap around）到 0xfffffffc。访存阶段中会发现这个异常。在同一周期中，addl 指令处于执行阶段，而它会将条件码设置成新的值。这就会违反异常点之后的所有指令都不能影响系统状态的要求。

一般地，通过将异常处理逻辑合并到流水线结构中，我们既能够从各个异常中做出正确的选择，也能够避免出现由于分支预测错误取出的指令造成的异常。我们给每个流水线寄存器添加了一个特殊的字段 exc，它给出处于该流水线寄存器中指令的异常状态。如果一条指令在其处理中于某个阶段产生了一个异常，状态字段就设置成指示异常的种类。异常状态和其他信息一起沿着流水线传播，直到它到达写回阶段。在此，流水线控制逻辑发现出异常，并开始取出异常处理程序的代码。

为了避免异常点之后的指令更新任何程序员可见的状态，应该修改流水线控制逻辑，使之在访存或写回阶段中的指令导致异常时，不会更新条件码寄存器或是数据存储器。在上面的示例程序中，控制逻辑会发现访存阶段中的 pushl 导致了异常，因此应该禁止 addl 指令更新条件码寄存器。（在本段文字所对应的 PIPE 的模拟器中，你会看到流水线化的处理器中处理异常的技术实现。）

让我们来看看这种处理异常的方法是怎样解决我们刚才提到的那些细节问题的。当流水线中有一个或多个阶段出现异常时，信息只是简单地存放在流水线寄存器的异常状态字段中。异常事件不会对流水线中的指令流有任何影响，除了会禁止流水线中后面的指令更新程序员可见的状态（条件码寄存器或存储器），直到异常指令到达最后的流水线阶段。因为指令到达写回阶段的顺序与它们在非流水线化的处理器中执行的顺序相同，所以我们可以保证第一条遇到异常的指令会第一个引起控制转移到异常处理程序。如果取出了某条指令，过后又取消了，那么所有关于这条指令的异常状态信息也都会被取消。所有导致异常的指令后面的指令都不能改变程序员可见的状态。携带指令的异常状态以及所有其他信息通过流水线的简单原则是处理异常的简单而可靠的机制。

多周期指令

Y86 指令集中的所有指令都包括一些简单的操作，例如数字加法。这些操作可以在执行阶段一个周期内处理完。在一个更完整的指令集中，我们还需要实现一些需要更为复杂操作的指令，例如，

整数乘法和除法，以及浮点运算。在一个像 PIPE 这样性能中等的处理器中，这些操作的典型执行时间从浮点加法的 3 或 4 个周期到整数除法的 32 个周期。为了实现这些指令，我们既需要额外的硬件来执行这些计算，还需要一种机制来协调这些指令的处理与流水线其他部分之间的关系。

实现多周期指令的一种简单方法就是只是简单地扩展执行阶段逻辑的功能，添加一些整数和浮点算术运算单元。一条指令在执行阶段中逗留它所需要的多个时钟周期，会导致取指和解码阶段暂停。这种方法实现起来很简单，但是得到的性能并不是太好。

可以通过采用独立于主流水线的特殊硬件功能单元来处理较为复杂的操作以得到更好的性能。通常，有一个功能单元来执行整数乘法和除法，还有一个来执行浮点操作。当一条指令进入解码阶段时，它可以被发射到特殊单元。在这个特殊单元执行该操作时，流水线会继续处理其他指令。通常，浮点单元本身也是流水线化的，因此多个操作可以在主流水线和各个单元中并发执行。

不同单元的操作必须同步，以避免出错。比如说，如果在被不同单元执行的各个指令之间有数据相关，控制逻辑可能需要暂停系统的某个部分，直到由系统其他某个部分处理的操作的结果完成。经常使用各种形式的转发，将结果从系统的一个部分传递到其他部分，和我们前面看到的 PIPE 各个阶段之间的转发一样。虽然与 PIPE 相比，整个设计变得更为复杂，但还是可以使用暂停、转发以及流水线控制等同样的技术来使整体行为与顺序的 ISA 模型相匹配。

存储系统的接口

在我们对 PIPE 的描述中，我们假设取指单元和数据存储器都可以在一个时钟周期内读或是写存储器中任意的位置。我们还忽略了由自我修改（self-moodifying）代码造成的可能冒险。在自我修改代码中，一条指令对一个存储区域进行写，而后面的指令又从这个区域中读取。进一步说，我们是以存储器位置的虚拟地址来引用它们的，这就要求在执行实际的读或写操作之前，要将虚拟地址翻译成物理地址。显然，要在一个时钟周期内完成所有这些处理是不现实的。更糟糕的是，正在访问的存储器的值可能是位于磁盘上的，这会需要上百万个时钟周期才能把数据读入到处理器存储器中。

正如我们将在第 6 章和第 10 章中讲述的那样，处理器的存储系统是由多种硬件存储器和管理虚拟存储器的操作系统软件共同组成的。存储系统被组织成一个层次结构，较快但是较小的存储器保持着存储器的一个子集，而较慢但是较大的存储器作为它的后备。最靠近处理器的一层是高速缓存存储器（cache memories），它提供对最常使用的存储器位置的快速访问。一个典型的处理器有两个第一层高速缓存——一个用于读指令，一个用于读和写数据。另一种类型的高速缓存存储器，称为翻译后备缓冲器（translation look-aside buffer）或 TLB，它提供了从虚拟地址到物理地址的快速翻译。将 TLB 和高速缓存结合起来使用，大多数时候，确实可能在一个时钟周期内读指令并读或是写数据。因此，对我们的处理器引用存储器的简化的看法实际上是很合理的。

虽然高速缓存中保存有最常引用的存储器位置[1]，但是还是有时候会出现高速缓存不命中，也就是有些引用的位置不在高速缓存中。最好的情况中，可以从较高层的高速缓存或处理器的主存中找到不命中的数据，这需要 3～20 个时钟周期。同时，流水线会暂停，将指令保持在取指或访存阶段，直到高速缓存能够执行读或写操作。至于我们的流水线设计，通过添加更多的暂停条件到流水线控制逻辑，就能实现这个功能。高速缓存不命中以及随之而来的与流水线的同步都完全是由硬件来处理的，这样能使所需的时间尽可能地缩短到很少数量的时钟周期。

---

1 指数据。——译者

有时，被引用的存储器位置实际上是存储在磁盘存储器上的。此时，硬件会产生一个缺页（page fault）异常信号。同其他异常一样，这个异常会导致处理器调用操作系统的异常处理程序代码。然后这段代码会发起从磁盘到主存的传送操作。一旦完成，操作系统会返回到原来的程序，而导致缺页的指令会被重新执行。这次存储器引用将成功，虽然可能会导致高速缓存不命中。让硬件调用操作系统例程，然后它又会将控制返回给硬件，这就使得硬件和系统软件在处理缺页时能协同工作。因为访问磁盘会需要数百万个时钟周期，OS 缺页中断处理程序执行的处理所需的几百个时钟周期对性能的影响可以忽略不计。

从处理器的角度来看，将用暂停来处理短时间的高速缓存不命中和用异常处理来处理长时间的缺页结合起来，能够顾及到存储器访问时由于存储器层次结构引起的所有不可预测性。

旁注：当前的微处理器设计

一个五阶段流水线，例如我们已经讲过的 PIPE 处理器，代表了 20 世纪 80 年代中期的处理器设计水平。Berkeley 的 Patterson 研究组开发的 RISC 处理器原型是第一个 SPARC 处理器的基础，它是 Sun Microsystems 在 1987 年开发的。Stanford 的 Hennessy 的研究组开发的处理器由 MIPS Technologies（一个由 Hennessy 成立的公司）在 1986 年商业化了。这两种处理器都使用的是五阶段流水线。Intel 的 i486 处理器用的也是五阶段流水线，只不过阶段之间的职责划分不太一样，它有两个解码阶段和一个合并了的执行/访存阶段[21]。

这些流水线化的设计的吞吐量都限制在最多一个时钟周期一条指令。4.5.10 小节中描述的 CPI（每指令周期）测量值不可能超过 1.0。不同的阶段一次只能处理一条指令。较新的处理器支持超标量（superscalar）操作，意味着它们通过并行地取指、解码和执行多条指令，可以实现小于 1.0 的 CPI。当超标量处理器已经广泛使用时，性能测量标准已经从 CPI 转化成了它的倒数——每周期执行指令的平均数，即 IPC。对超标量处理器来说，IPC 可以大于 1.0。最先进的设计使用了一种称为乱序（out-of-order）执行的技术来并行地执行多条指令，执行的顺序也可能完全不同于它们在程序中出现的顺序，但是保留了顺序 ISA 模型蕴含的整体行为。作为对程序优化的讨论的一部分，我们将会在第 5 章中讨论这种形式的执行。

不过，流水线化的处理器并不只有传统的用途。现在出售的大部分处理器都用在嵌入式系统中，控制着汽车运行、消费产品，以及其他一些系统用户不能直接看到处理器的地方。在这些应用中，与性能较高的模型相比，流水线化的处理器的简单性，比如说像我们在本章中讨论的这样，会降低成本和功耗需求。

4.6 小结

我们已经看到，指令集体系结构（即 ISA）在处理器行为（就指令集合及其编码而言）和如何实现处理器之间提供了一层抽象。ISA 提供了程序执行的一种顺序说明，也就是一条指令执行完了，下一条指令才会开始。

基本 IA32 指令集，并且大大简化其数据类型、地址模式和指令编码，我们定义出了 Y86 指令集。得到的 ISA 既有 RISC 指令集的属性，也有 CISC 指令集的属性。然后，我们将不同指令组织

放到五[2]个阶段中处理，在此，根据被执行的指令的不同，每个阶段中的操作也不相同。从此，我们构造了 SEQ 处理器，其中每个时钟周期推进一条指令通过每个阶段。通过重新排列各个阶段，我们创建了 SEQ+设计，其中第一个阶段选择程序计数器的值，它被用来取出当前指令。

流水线化通过让不同的阶段并行操作，改进了系统的吞吐量性能。在任意一个给定的时刻，多条指令被处理。在引入这种并行性的过程中，我们必须非常小心，以提供与程序的顺序执行相同的用户可见的、程序级行为。我们通过在 SEQ+中添加流水线寄存器，并重新安排周期来创建 PIPE-流水线，介绍了流水线化。然后，我们添加了转发逻辑，加速了将结果从一条指令发送到另一条指令，从而提高了流水线的性能。有几种特殊情况需要额外的流水线控制逻辑来暂停或取消一些流水线阶段。

在本章中，我们学习了有关处理器设计的几个重要经验：

- **管理复杂性是首要问题。**我们想要优化使用硬件资源，在最小的成本下获得最大的性能。为了实现这个目的，我们创建了一个非常简单而一致的框架，来处理所有不同的指令类型。有了这个框架，我们就能够在处理不同指令类型的逻辑中间共享硬件单元。
- **我们不需要直接实现 ISA。**ISA 的直接实现意味着一个顺序的设计。为了获得更高的性能，我们想运用硬件能力以同时执行许多操作，这就导致要使用流水线化的设计。通过仔细的设计和分析，我们能够处理各种流水线冒险，因此运行一个程序的整体效果，同用 ISA 模型获得的效果完全一致。
- **硬件设计人员必须非常谨慎小心。**一旦芯片被制造出来，就几乎不可能改正任何错误了。一开始就使设计正确是非常重要的。意思就是，仔细地分析各种指令类型和组合情况，甚至于那些看上去没有意义的情况，例如弹出栈指针。必须用系统的模拟测试程序彻底地测试设计。在开发 PIPE 的控制逻辑中，我们的设计有个细微的错误，只有通过对控制组合的仔细而系统的分析才能发现。

4.6.1 Y86 模拟器

本章的实验资料包括 SEQ、SEQ+和 PIPE 处理器的模拟器。每个模拟器都有两个版本：

- GUI（图形用户界面）版本在图形窗口中显示存储器、程序代码以及处理器状态。它提供了一种查看指令如何通过处理器的方便形式。控制面板还允许你交互式地重启动、单步或运行模拟器。这些版本要求有 Tcl 脚本语言和 Tk 图形库。
- 文本版本运行的是相同的模拟器，但是它只将显示信息打印到终端上。对调试来讲，这个版本不是很有用，但是它允许处理器的自动测试，而且它可以运行在不支持 Tcl/Tk 的系统上。

模拟器的控制逻辑是通过将逻辑块的 HCL 声明翻译成 C 代码产生的。然后，将该代码编译并与模拟代码的其他部分进行链接。同时还有测试脚本，它们全面地测试各种指令以及各种冒险的可能性。

参考文献说明

对于那些想更多地学习逻辑设计的人来说，Katz 的逻辑设计教科书[39]是标准的入门教材，它强调的是硬件描述语言的使用。

Hennessy 和 Patterson 的计算机体系结构教科书[33]覆盖了处理器设计的广泛内容，包括像我们

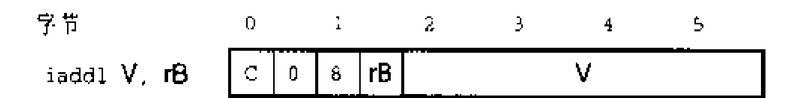
2 原文是六。——译者

在这里讲述的简单流水线，还有并行执行更多指令的更高级的处理器。Shriver 和 Smith[69]详细介绍了 AMD 制造的、Intel 兼容的 IA32 处理器。

家庭作业

4.32 ◆

在我们的 Y86 示例程序中，例如图 4.5 中的 Sum 函数，我们多次遇到想将一个常数加到寄存器的情况（例如，第 12 和 13 行，以及第 14 和 15 行）。这要求首先用 irmovl 指令将一个寄存器设置为常数，然后用 addl 指令把这个值加到目的寄存器。假设我们想添加一条新指令 iaddl，其格式如下：



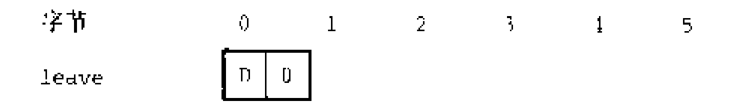
这条指令将常数值 V 加到寄存器 rB。请描述实现这一指令所执行的计算。可以参考 irmovl 和 OPl 的计算（图 4.16）。

4.33 ◆

如 3.7.2 小节中讲述的那样，IA32 的指令 leave 可以用来使栈为返回做准备。它等价于下面这个 Y86 指令序列：

```
1   rrmovl  %ebp, %esp  Set stack pointer to beginning of frame
2   popl    %ebp        Restore saved %ebp and set stack ptr to end of
                        caller's frame
```

假设我们要往 Y86 指令集中加入这样一条指令，编码如下：



请描述实现这一指令所执行的计算。可以参考 popl 的计算（图 4.18）。

4.34 ◆◆

文件 seq-full.hcl 包含 SEQ 的 HCL 描述，并将常数 IIADDL 声明为十六进制值 C，也就是 iaddl 的指令代码。修改实现 iaddl 指令的控制逻辑块的 HCL 描述，就像家庭作业 4.32 中描述的那样。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.35 ◆◆

文件 seq-full.hcl 还将常数 ILEAVE 声明为十六进制值 D，也就是 leave 的指令代码，同时将常数 REBP 声明为 7，即%ebp 的寄存器 ID。修改实现 leave 指令的控制逻辑块的 HCL 描述，就像家庭作业 4.33 中描述的那样。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.36 ◆◆◆

假设我们要创建一个较低成本的、基于我们为 PIPE- 设计的结构（图 4.39 和图 4.41）的流水线化的处理器，没有使用旁路技术。这个设计用暂停来处理所有的数据相关，直到产生所需值的指令已经通过了写回阶段。

文件 pipe-stall.hcl 包含一个对 PIPE 的 HCL 代码的修改版，其中禁止了旁路逻辑。也就是，信号 e_valA 和 e_valB 只是简单地声明为下面这样：

```
## DO NOT MODIFY THE FOLLOWING CODE.
## No forwarding.  valA is either valP or value from register file
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        1 : d_rvalA; # Use value read from register file
];

## No forwarding. valB is value from register file
int new_E_valB = d_rvalB;
```

修改文件结尾处的流水线控制逻辑，使之能正确处理所有可能的控制和数据冒险。作为设计工作的一部分，你还要分析各种控制情况的组合，就像我们在 PIPE 的流水线控制逻辑设计中做的那样。你会发现有许多不同的组合，因为有更多的情况需要流水线暂停。要确保你的控制逻辑能正确处理每种组合情况。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.37 ◆◆

文件 pipe-full.hcl 包含一份 PIPE 的 HCL 描述，以及常数值 IIADDL 的声明。修改该文件以实现指令 iaddl，就像家庭作业 4.32 中描述的那样。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.38 ◆◆◆

文件 pipe-full.hcl 还包含常数 ILEAVE 和 REBP 的声明。修改该文件以实现指令 leave，就像家庭作业 4.33 中描述的那样。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.39 ◆◆◆

文件 pipe-nt.hcl 包含一份 PIPE 的 HCL 描述，并将常数 J_YES 声明为值 0，即无条件转移指令的函数代码。修改分支预测逻辑，使之对条件转移预测为不选择分支，而对无条件转移和 call 预测为选择分支。你需要设计一种方法来得到跳转目标地址 valC，并送到流水线寄存器 M，以便从错误的分支预测中恢复。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.40 ◆◆◆

文件 pipe-btfnt.hcl 包含一份 PIPE 的 HCL 描述，并将常数 J_YES 声明为值 0，即无条件转移指令的函数代码。修改分支预测逻辑，使得当 valC < valP 时（后向分支），就预测条件转移为选择分支，当 valC ≥ valP 时（前向分支），就预测为不选择分支。并且将无条件转移和 call 预测为选择分支。你需要设计一种方法来得到 valC 和 valP，并送到流水线寄存器 M，以便从错误的分支预测中恢复。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.41 ◆◆◆

在我们的 PIPE 的设计中，只要一条指令执行了 load 操作，从存储器中读一个值到寄存器，并且下一条指令要用这个寄存器作为源操作数，就会产生一个暂停。如果要在执行阶段中使用这个源操作数，暂停是避免冒险的惟一方法。

对于第二条指令将源操作数存储到存储器的情况，例如 rmmovl 或 pushl 指令，是不需要这样的暂停的。考虑下面这段代码示例：

```
1    mrmovl 0(%ecx),%edx      # Load 1
2    pushl %edx               # Store 1
3    nop
4    popl %edx                # Load 2
5    rmmovl %eax,0(%edx)      # Store 2
```

在第 1 行和第 2 行，mrmovl 指令从存储器读一个值到%edx，然后 pushl 指令将这个值压入栈中。我们的 PIPE 设计会让 pushl 指令暂停，以避免装载/使用冒险。不过，可以看到，pushl 指令要到访存阶段才会需要%edx 的值。我们可以再添加一条旁路通路，如图 4.69 所示，将存储器输出（信号 m_valM）转发到流水线寄存器 M 中的 valA 字段。在下一个时钟周期，被传送的值就能写入存储器了。这种技术称为*加载转发*（load forwarding）。

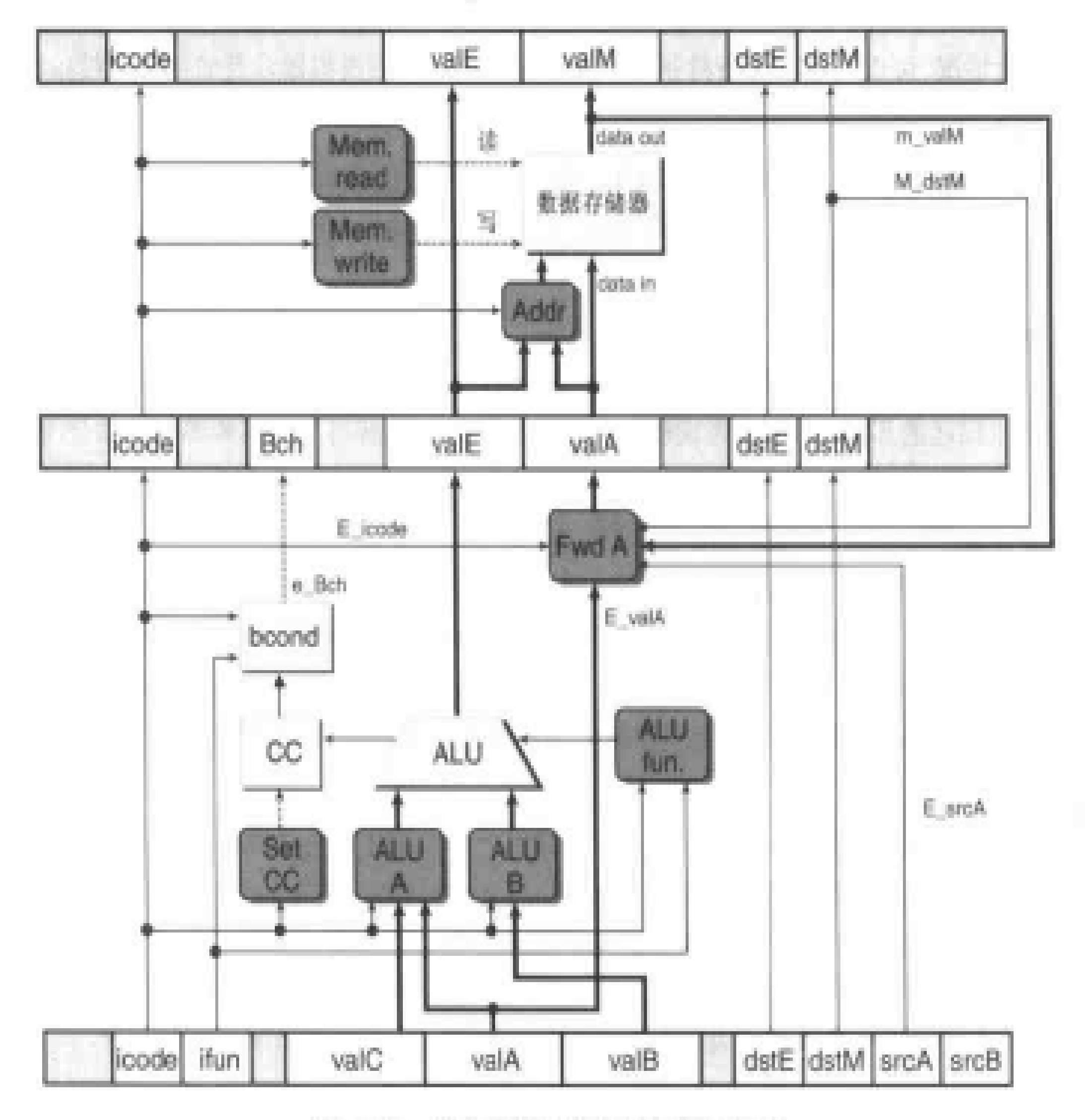


图 4.69 带加载转发的执行和访存阶段

通过添加一条从存储器输出到流水线寄存器 M 中 valA 的源的旁路通路，对于这种形式的加载/使用冒险，我们可以使用转发而不必暂停。这是家庭作业 4.41 的主题。

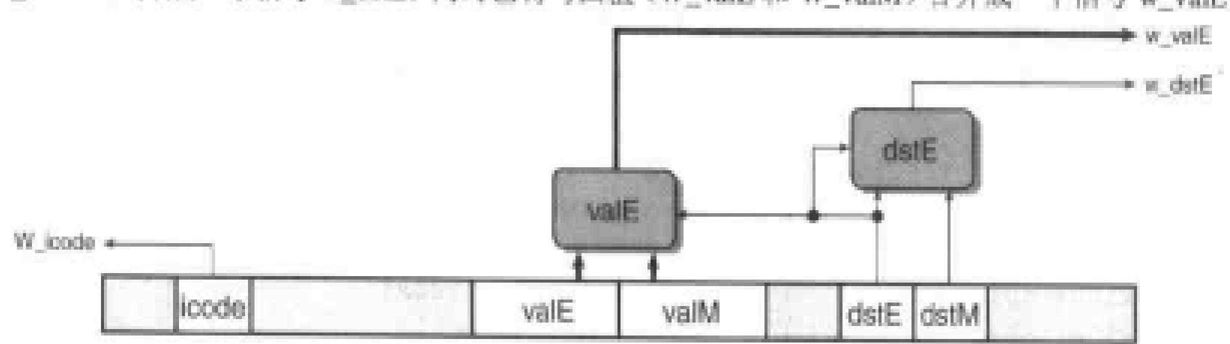
注意，上述代码序列中的第二个例子（第4行和第5行）不能利用加载转发。popl 指令加载的值是作为下一条指令地址计算的一部分的，而在执行阶段而非访存阶段就需要这个值了。

A. 写出描述发现加载/使用冒险条件的公式，类似于图4.64中给出的那一个，除了在能用加载转发时不会导致暂停以外。

B. 文件 pipe-lf.hcl 包含一个 PIPE 控制逻辑的修改版。它含有信号 new_M_valA 的定义，是用来实现图4.69中标号为“Fwd A”的块的。它还将流水线控制逻辑中的加载/使用冒险的条件设置为0，因此流水线控制逻辑不会发现任何形式的加载/使用冒险。修改这个 HCL 描述以实现加载转发。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

4.42 ◆◆◆

我们的流水线化的设计有点不太现实，因为寄存器文件有两个写端口，然而只有 popl 指令需要对寄存器文件同时进行两个写操作。因此，其他指令只使用一个写端口，共享这个端口来写 valE 和 valM。下面这个图给出的是一个对写回逻辑的修改版，其中，我们将写回寄存器 ID（W_dstE 和 W_dstM）合并成一个信号 w_dstE，同时也将写回值（W_valE 和 W_valM）合并成一个信号 w_valE：



用 HCL 写的执行这些合并的逻辑，如下所示：

```
int w_dstE = [
        ## writing from valM
        W_dstM != RNONE : W_dstM;
        1: W_dstE;
];
int w_valE = [
        W_dstM != RNONE : W_valM;
        1: W_valE;
];
```

对这些多路复用器的控制是由 dstE 确定的——当它表明有某个寄存器时，就选择端口 E 的值，否则就选择端口 M 的值。

在模拟模型中，我们可以禁止寄存器端口 M，如下面这段 HCL 代码所示：

```
int w_dstM = RNONE;
int w_valM = 0;
```

接下来的问题就是要设计处理 popl 的方法。一种方法是用控制逻辑动态地处理指令 popl rA，使之与下面两条指令序列有一样的效果：

```
iaddl $4, %esp
mrmovl -4(%esp), rA
```

（关于指令 iaddl 的描述，请参考家庭作业 4.32）要注意两条指令的顺序，以保证 popl %esp 能正确工作。要达到这个目的，可以让解码阶段的逻辑对上面列出的 popl 指令和 addl 指令一视同仁，除了它会预测下一个 PC 与当前 PC 相等以外。在下一个周期，再次取出了 popl 指令，但是指令代码变成了特殊的值 IPOP2。它会被当作一条特殊的指令来处理，行为与上面列出的 mrmovl 指令一样。

文件 pipe-lw.hcl 包含着上面讲的修改过的写端口逻辑。它将常数 IPOP2 声明为十六进制值 E。还包括信号 new_D_icode 的定义，它产生流水线寄存器 D 的 icode 字段。可以修改这个定义，使得当第二次取出 popl 指令时，插入这个指令代码。这个 HCL 文件还包含信号 f_pc 的声明，也就是标号为“Select PC”的块（图 4.56）在取指阶段产生的程序计数器的值。

修改该文件中的控制逻辑，使之按照我们描述的方式来处理 popl 指令。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

练习题答案

练习题 4.1 答案

手工对指令编码是非常乏味的，但是它将巩固你对汇编器将汇编代码变成字节序列的理解。在下面这段我们的 Y86 汇编器的输出中，每一行都给出了一个地址和一个从该地址开始的字节序列。

```
1 0x100:                  | .pos 0x100 # Start generating code at address 0x100
2 0x100: 30830f000000     |     irmovl $15,%ebx
3 0x106: 2031             |     rrmovl %ebx,%ecx
4 0x108:                  | loop:
5 0x108: 40135dffffff     |     rmmovl %ecx,-3(%ebx)
6 0x10e: 6031             |     addl   %ebx,%ecx
7 0x110: 7008010000       |     jmp loop
```

这段编码有些地方值得注意：

- 十进制的 15（第 2 行）的十六进制表示为 0x0000000f。以反向顺序来写就是 0f 00 00 00。
- 十进制-3（第 5 行）的十六进制表示为 0xfffffffd。以反向顺序来写就 fd ff ff ff。
- 代码从地址 0x100 开始。第一条指令需要 6 个字节，而第二条需要 2 个字节。因此，循环的目标地址为 0x00000108。以反向顺序来写就是 08 01 00 00。

练习题 4.2 答案

手工对一个字节序列进行解码能帮助你理解处理器面临的任务。它必须读入字节序列，并确定该执行什么指令。接下来，我们给出的是用来产生每个字节序列的汇编代码。在汇编代码的左边，你可以看到每条指令的地址和字节序列。

A. 带立即数和地址位移的操作。

```
0x100: 3083fcffffff |   irmovl $-4,%ebx
0x106: 406300080000 |   rmmovl %esi,0x800(%ebx)
0x10c: 10           |   halt
```

B. 包含一个函数调用的代码。

```
0x200: a068         |   pushl %esi
0x202: 8008020000   |   call proc
```

```
0x207: 10           |    halt
0x208:              |proc:
0x208: 30830a000000 |    irmovl $10,%ebx
0x20e: 90           |    ret
```

C. 包含非法指令指示字节 0xf0 的代码。

```
0x300: 505407000000 |    mrmovl 7(%esp),%ebp
0x306: 00           |    nop
0x307: f0           | .byte 0xf0 # invalid instruction code
0x308: b018         |    popl %ecx
```

D. 包含一个跳转操作的代码。

```
0x400:              | loop:
0x400: 6113         |    subl %ecx, %ebx
0x402: 7300040000   |    je loop
0x407: 10           |    halt
```

E. pushl 指令中第二个字节为非法的代码。

```
0x500: 6362         |    xorl %esi,%edx
0x502: a0           |    .byte 0xa0 # pushl instruction code
0x503: 80           |    .byte 0x80 # Invalid register byte
```

练习题 4.3 答案

正如题目中建议的那样，我们修改了 IA32 机器上的 GCC 产生的代码：

```
        # int Sum(int *Start, int Count)
rSum:   pushl %ebp
        rrmovl %esp,%ebp
        irmovl $20,%eax
        subl %eax,%esp
        pushl %ebx
        mrmovl 8(%ebp),%ebx
        mrmovl 12(%ebp),%eax
        andl %eax,%eax
        jle L38
        irmovl $-8,%edx
        addl %edx,%esp
        irmovl $-1,%edx
        addl %edx,%eax
        pushl %eax
        irmovl $4,%edx
        rrmovl %ebx,%eax
        addl %edx,%eax
        pushl %eax
        call rSum
        mrmovl (%ebx),%edx
        addl %edx,%eax
        jmp L39
L38:    xorl %eax,%eax
```

```
L39:    mrmovl -24(%ebp),%ebx
        rrmovl %ebp,%esp
        popl %ebp
        ret
```

练习题 4.4 答案

虽然很难想像这条指令有什么实际用处，但是在设计系统时，避免在这种情况中出现歧义是非常重要的。我们要为这条指令的行为确定一种合理的解释惯例，并确保每个实现都遵守了这个惯例。

这个测试中 subl 指令比较了%esp 的初始值和压入栈中的值。相减的结果为 0，这表明压入栈中的是%esp 原来的值。

练习题 4.5 答案

更难以想像会有人想要将栈顶值弹出到栈指针中。不过，我们还是要确定一个惯例并遵循它。这个代码序列将 tval 压入栈中，再弹出到%esp 中，并返回弹出的值。既然结果等于 tval，我们可以推断出 popl %esp 应该是将栈指针设置为从存储器中读出的值。因此，它等价于指令 `mrmovl 0(%esp),%esp`。

练习题 4.6 答案

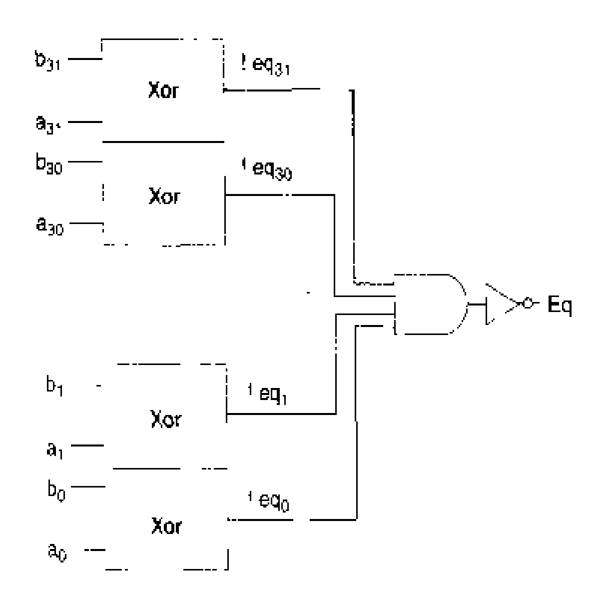
EXCLUSIVE-OR（异或）函数要求两个位有相反的值：

```
bool eq = (!a && b) || (a && !b);
```

通常，信号 eq 和 xor 是互补的。也就是，一个等于 1，另一个就等于 0。

练习题 4.7 答案

EXCLUSIVE-OR 电路的输出是位相等值的补。根据德摩根定律（图 2.7），我们能用 OR 和 NOT 实现 AND，得到如下电路：



练习题 4.8 答案

这个设计只是对从三个输入中找出最小值的设计做了点简单的改变。

```
int Med3 = [
```

```
        A <= B && B <= C  : B;
        B <= A && A <= C  : A;
        1                 : C;
];
```

练习题 4.9 答案

这些练习使各个阶段的计算更加具体。从目标代码中我们可以看到，指令是位于地址 0x00e 的。它包含 6 个字节，前两个字节为 0x30 和 0x84。后四个字节是 0x00000080（十进制 128）按字节反过来的形式。

阶段	通用	具体
	irmovl V, rB	irmovl $128, %esp
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:ifun ← M_1[0x00e]=3:0 rA:rB ← M_1[0x00f]=8:4 valC ← M_4[0x010]=128 valP ← 0x00e+6=0x014
解码		
执行	valE ← 0+valC	valE ← 0+128=128
访问		
写回	R[rB] ← valE	R[%esp] ← valE=128
更新 PC	PC ← valP	PC ← valP = 0x014

这个指令将寄存器%esp 设为 128，并将 PC 加 6。

练习题 4.10 答案

我们可以看到指令位于地址 0x01c，有两个字节，值分别为 0xb0 和 0x08。pushl 指令（第 6 行）将寄存器%esp 设为了 124，存储器中该位置存储着的值为 9。

阶段	通用	具体
	popl rA	popl %eax
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[0x01c]=b:0 rA:rB ← M_1[0x01d]=0:8 valP ← 0x01c+2=0x01e
解码	valA ← R[%esp] valB ← R[%esp]	valA ← R[%esp]=124 valB ← R[%esp]=124
执行	valE ← valB+4	valE ← 124+4=128
访存	valM ← M_4[valA]	valM ← M_4[124]=9
写回	R[%esp] ← valE R[rA] ← valM	R[%esp] ← 128 R[%esp] ← 9
更新 PC	PC ← valP	PC ← 0x01e

该指令将%eax 设为 9，将%esp 设为 128，并将 PC 加 2。

练习题 4.11 答案

沿着图 4.18 中列出的步骤，将 rA 看成%esp，我们可以看到，在访存阶段，指令会将 valA（即栈指针的原始值）存放到存储器中，与我们在 IA32 中发现的一样。

练习题 4.12 答案

沿着图 4.18 中列出的步骤，将 rA 看成%esp，我们可以看到，两个写回操作都会更新%esp。因为写 valM 的操作后发生，指令的最终效果会是将从存储器中读出的值写入%esp，就像在 IA32 中看到的一样。

练习题 4.13 答案

我们可以看到这条指令位于地址 0x023，长度为 5 个字节。第一个字节值为 0x80，而后面 4 个字节是 0x00000029，即调用的目标地址按字节反过的形式。popl 指令（第 7 行）将栈指针设为 128。

阶段	通用	具体
	call Dest	call 0x029
取指	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	icode:ifun ← M_1[0x023]=8:0 valC ← M_4[0x024]=0x029 valP ← 0x023+5=0x028
解码	 valB ← R[%esp]	 valB ← R[%esp]=128
执行	valE ← valB+-4	valE ← 128+-4=124
访存	M_4[valE] ← valP	M_4[124] ← 0x028
写回	R[%esp] ← valE	R[%esp] ← 124
更新 PC	PC ← valC	PC ← 0x029

这条指令的效果就是将%esp 设为 124，将 0x028（返回地址）存放到该存储器地址，并将 PC 设为 0x029（调用的目标地址）。

练习题 4.14 答案

练习题中所有的 HCL 代码都很简单明了，但是试着自己写会帮助你思考各个指令，以及如何处理它们。对于这个问题，我们只要看看 Y86 的指令集（图 4.2），确定哪些有常数字段。

```
bool need_valC =
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
```

练习题 4.15 答案

这段代码类似于 srcA 的代码：

```
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

练习题 4.16 答案

这段代码类似于 dstE 的代码：

```
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        1 : RNONE; # Don't need register
];
```

练习题 4.17 答案

像我们在练习题 4.12 中发现的那样，为了将从存储器中读出的值存放到%esp，我们想让通过 M 端口写的优先级高于通过 E 端口写。

练习题 4.18 答案

这段代码类似于 aluA 的代码：

```
int aluB = [
        icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                        IPUSHL, IRET, IPOPL } : valB;
        icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];
```

练习题 4.19 答案

这段代码类似于 mem_addr 的代码：

```
int mem_data = [
        # Value from register
        icode in { IRMMOVL, IPUSHL } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];
```

练习题 4.20 答案

这段代码类似于 mem_read 的代码：

```
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
```

练习题 4.21 答案

这个题目是个非常有趣的练习，它试图在一组划分中找到优化平衡。它要求计算许多流水线的吞吐量和执行时间。

A. 对一个两阶段流水线来说，最好的划分是块 A、B 和 C 在第一阶段，块 D、E 和 F 在第二阶段。第一阶段的延迟为 170ps，所以整个周期的时长为 170+20=190ps。因此吞吐量为 5.26 GOPS，而执行时间为 380ps。

B. 对一个三阶段流水线来说，应该使块 A 和 B 在第一阶段，块 C 和 D 在第二阶段，而块 E 和 F 在第三阶段。前两个阶段的延迟均为 110ps，所以整个周期时长为 130ps，而吞吐量为 7.69 GOPS。执行时间为 390ps。

C. 对一个四阶段流水线来说，块 A 为第一阶段，块 B 和 C 在第二阶段，块 D 是第三阶段，而块 E 和 F 在第四阶段。第二阶段需要 90ps，所以整个周期时长为 110ps，而吞吐量为 9.09 GOPS。执行时间为 440ps。

D. 最优的设计应该是五阶段流水线，除了 E 和 F 处于第五阶段以外，其他每个块是一个阶段。周期时长为 80+20=100ps，吞吐量为大约 10.00 GOPS，而执行时间为 500ps。变成更多的阶段也不会有帮助了，因为不可能使流水线运行得比以 100ps 为一周期还要快了。

练习题 4.22 答案

在这种极限情况下，流水线的每个计算块的延迟都为 ε ns。时钟周期为 ε+20 ps，吞吐量为 1000/(ε+20)。如果阶段数量变成任意大，ε 会趋向于 0，因此吞吐量为 50.00 GOPS。

练习题 4.23 答案

这段代码只是给 SEQ 代码中的信号名前加上前缀“D_.”。

```
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

练习题 4.24 答案

由于 popl 指令（第 4 行）造成的加载/使用冒险，rrmovl 指令（第 5 行）会暂停一个周期。当它进入解码阶段，popl 指令处于访存阶段，使 M.dstE 和 M.dstM 都等于%esp。如果两种情况反过来，那么来自 M_valE 的写回优先级较高，导致增加了的栈指针被传送到 rrmovl 指令作为参数。这与练习题 4.5 中确定的处理 popl %esp 的惯例不一致。

练习题 4.25 答案

这个问题让你体验一下处理器设计中一个很重要的任务——为一个新处理器设计测试程序。通常，我们的测试程序应该能测试所有的冒险可能性，而且一旦有相关不能被正确处理，就会产生错误的结果。

对于此例，我们可以使用对练习题 4.24 中所示的程序稍微修改了一点的版本：

```
1    irmovl $5, %edx
2    irmovl $0x100,%esp
3    rmmovl %edx,0(%esp)
4    popl %esp
5    nop
6    nop
7    rrmovl %esp,%eax
```

两个 nop 指令会导致当 rrmovl 指令在解码阶段中时，popl 指令处于写回阶段。如果给予处于写回阶段中的两个转发源错误的优先级，那么寄存器%eax 会设置成增加了的程序计数器，而不是从存储器中读出的值。

练习题 4.26 答案

这个逻辑只需要检查五个转发源：

```
int new_E_valB = [
        d_srcB == E_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;    # Use value read from register file
];
```

练习题 4.27 答案

下面这个测试程序是设计用来建立控制组合 A（图 4.67），并探测是否出了错：

```
1   # Code to generate a combination of not-taken branch and ret
2            irmovl Stack, %esp
3            irmovl rtnp,%eax
4            pushl %eax          # Set up return pointer
5            xorl %eax,%eax      # Set Z condition code
6            jne target          # Not taken (First part of combination)
7            irmovl $1,%eax      # Should execute this
8            halt
9   target:  ret                 # Second part of combination
10           irmovl $2,%ebx      # Should not execute this
11           halt
12  rtnp:    irmovl $3,%edx      # Should not execute this
13           halt
```

```
14    .pos 0x40
15    Stack:
```

设计这个程序是为了出错（例如实际上执行了 ret 指令）时，程序会执行一条额外的 irmovl 指令，然后停止。因此，流水线中的错误会导致某个寄存器更新错误。这段代码说明实现测试程序需要非常小心。它必须建立起可能的错误条件，然后再探测是否有错误发生。

练习题 4.28 答案

下面这个测试程序是设计用来建立控制组合 B（图 4.67）的。模拟器会发现流水线寄存器的气泡和暂停控制信号都设置成 0 的情况，因此我们的测试程序只需要建立它需要发现的组合情况。最大的挑战在于当处理正确时，程序要做正确的事情。

```
1     # Test instruction that modifies %esp followed by ret
2             irmovl mem,%ebx
3             mrmovl 0(%ebx),%esp   # Sets %esp to point to return point
4             ret                   # Returns to return point
5             halt                  #
6     rtnpt: irmovl $5,%esi         # Return point
7             halt
8     .pos 0x40
9     mem:    .long stack           # Holds desired stack pointer
10    .pos 0x50
11    stack:    .long rtnpt         # Top of stack: Holds return point
```

这个程序使用了存储器中两个初始化了的字。第一个字（mem）保存着第二个字（stack——期望的栈指针）的地址。第二个字保存着 ret 指令期望的返回点的地址。这个程序将栈指针加载到%esp，并执行 ret 指令。

练习题 4.29 答案

从图 4.66 我们可以看到，由于加载/使用冒险，流水线寄存器 D 必须暂停。

```
bool D_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB };
```

练习题 4.30 答案

从图 4.66 中我们可以看到，由于加载/使用冒险，或者由于分支预测错误，流水线寄存器 E 必须设置成气泡：

```
bool E_bubble =
        # Mispredicted branch
```

```
(E_icode == IJXX && !e_Bch) ||
# Conditions for a load/use hazard
E_icode in { IMRMOVL, IPOPL } &&
   E_dstM in { d_srcA, d_srcB};
```

练习题 4.31 答案

此时，预测错误的频率是 0.35，得到 mp=0.20×0.35×2=0.14，而整个 CPI 为 1.25。看上去收获非常小，但是如果实现新的分支预测策略的成本不是很高的话，这样做还是值得的。

CHAPTER 5

优化程序性能

编写高效程序需要两类活动：第一，我们必须选择一组最好的算法和数据结构；第二，我们必须编写出编译器能够有效优化以转换成高效可执行代码的源代码。对于这第二部分，理解优化编译器的能力和局限性是很重要的。就我们所知，编写程序方式中一点小小的变动，都会引起编译器优化方式很大的变化。有些编程语言比其他语言容易优化得多。C 的有些特性，例如执行指针运算和强制类型转换的能力，使得对它优化很困难。程序员经常能够以一种使编译器更容易产生高效代码的方式来编写他们的程序。

在程序开发和优化的过程中，我们必须考虑代码使用的方式，以及影响它的关键因素。通常，程序员必须在实现和维护程序的简单性与它的运行速度之间做出权衡折衷。在算法级上，几分钟就能编写一个简单的插入排序，而一个高效的排序算法程序可能需要一天或更长的时间来实现和优化。在代码级上，许多低级别的优化往往会降低程序的可读性和模块性，使得程序容易出错，更难以修改或扩展。对于一个只会运行一次以产生一组数据点的程序，以一种尽量减少编程工作量并保证正确性的方式来编写程序就更为重要一些。对于会在性能非常重要的环境中反复执行的代码，例如网络路由器，通常更广泛的优化会适当一些。

在本章中，我们描述许多提高代码性能的技术。理想的情况是，编译器能够接受我们编写的任何代码，并产生尽可能高效的、具有指定行为的机器级程序。事实上，编译器只能执行有限的程序转换，而且妨碍优化的因素（optimization blocker）还会阻碍这种优化，妨碍优化的因素就是程序行为中那些严重依赖于执行环境的方面。程序员必须编写容易优化的代码，以帮助编译器。就编译器来说，编译技术被分为“与机器无关”和“与机器有关”两类。“与机器无关”的意思是，使用这些技术时可以不考虑将执行代码的计算机的特性，而“与机器有关”是指，这些技术是依赖于许多机器的低级细节的。我们的讲述也沿用了类似的顺序，先讲编写任何程序时都要执行的标准程序转换，然后讲效率依赖于目标机器和编译器特性的转换。这些转换通常还会降低代码的模块性和可读性，因此，应该在获得最大性能是首要目标时，才使用这些技术。

为了使程序性能最大化，程序员和编译器需要一个目标机器的模型，指明如何处理指令，以及各个操作的时序特性。例如，编译器必须知道时序信息，才能够确定是需要一条乘法指令，还是移位和加法的某种组合。现代计算机用复杂的技术来处理机器级程序，并行执行许多指令，而且执行顺序还可能不同于它们在程序中出现的顺序。程序员必须理解为了获得最大的速度，这些处理器是如何工作来调整程序的。基于 Intel 处理器的最新模型，我们提出了一个这种机器的高级模型。我们还设计了一种图形表示法，可以用来使处理器执行指令形象化，并且还可以预测程序性能。

我们以对优化大型程序的问题的讨论来结束这一章。我们描述了代码剖析程序（profilers）的使用，代码剖析程序是测量程序各个部分性能的工具。这种分析能够帮助找到代码中低效率的地方，并且确定程序中我们应该着重优化的部分。最后，我们给出了一个重要的观察结论（称为 Amdahl 定律），它量化了对系统某个部分进行优化所带来的整体效果。

在本章的描述中，我们使得代码优化看起来像按照某种特殊顺序，对代码进行一系列转换的简单线性过程。实际上，这项工作远非这么简单。需要相当多的试错法试验。当我们进行到后面的优化阶段时，这种方法尤其有用，到那时，看上去很小的变化会导致性能上很大的变化。相反，一些很有希望的技术被证明是无效的。正如我们在后面的例子中看到的那样，要确切解释为什么某段代码序列有某个执行时间，是很困难的。性能可能依赖于处理器设计的许多详细特性，而对此我们所知甚少。这也是我们尝试各种技术的变形和组合的另一个原因。

研究汇编代码是理解编译器以及产生的代码会如何运行的最有效的手段之一。仔细研究内循环的代码是一个很好的开端。人们可以确认降低性能的属性，例如过多的存储器（memory）引用和对寄存器不正确的使用。从汇编代码开始，我们甚至可以预测什么操作会并行执行，以及它们使用处理器资源的效率如何。

5.1 优化编译器的能力和局限性

现代编译器运用复杂精细的算法来确定一个程序中计算的是什么值，以及它们是被如何使用的。然后它们会利用一些机会来简化表达式，也就是在几个不同的地方使用一个计算，以降低一个给定的计算必须被执行的次数。编译器优化程序的能力受几个因素限制，包括：要求它们绝不能改变正确的程序行为；它们对程序行为、对使用它们的环境了解有限；需要很快地完成编译工作。

编译器优化对用户来说应该是不可见的。当程序员用优化选项（例如，使用-O 命令行选项）编译代码时，代码的行为应该和不带优化编译得到的代码行为完全一样，除了它应该运行得更快一点以外。这样的要求使得编译器不能使用某些类型的优化。

例如，考虑下面这两个过程：

```
1    void twiddle1(int *xp, int *yp)
2    {
3        *xp += *yp;
4        *xp += *yp;
5    }
6
7    void twiddle2(int *xp, int *yp)
8    {
9        *xp += 2* *yp;
10   }
```

乍一看，这两个过程似乎有相同的行为。它们都是将存储在由指针 yp 指示的位置处的值两次加到指针 xp 指示的位置处的值。另一方面，函数 twiddle2 效率更高一些。它只要求三次存储器引用（读*xp，读*yp，写*xp），而 twiddle1 需要六次（两次读*xp，两次读*yp，两次写*xp）。因此，如果要编译器编译过程 twiddle1，我们会认为基于 twiddle2 执行的计算能产生更有效的代码。

不过，考虑一下 xp 等于 yp 的情况。此时，函数 twiddle1 会执行下面的计算：

```
3 *xp += *xp; /* Double value at xp */
4 *xp += *xp; /* Double value at xp */
```

结果会是 xp 的值增加 4 倍。另一方面，函数 twiddle2 会执行下面的计算：

```
9 *xp += 2* *xp; /* Triple value at xp */
```

结果会是 xp 的值增加 3 倍。编译器不知道 twiddle1 会被如何调用，因此它必须假设参数 xp 和 yp 可能会相等。因此，它不能产生 twiddle2 风格的代码作为 twiddle1 的优化版本。

这个现象称为*存储器别名使用*（memory aliasing）。编译器必须假设不同的指针可能会指向存储器中同一个位置。这造成了一个主要的*妨碍优化的因素*，这也是可能严重限制编译器产生优化代码机会的程序的一个方面。

练习题 5.1

下面的问题说明了存储器别名使用可能会导致意想不到的程序行为的方式。考虑下面这个交换两个值的过程：

```
1   /* Swap value x at xp with value y at yp */
2   void swap(int *xp, int *yp)
3   {
4       *xp = *xp + *yp; /* x+y */
5       *yp = *xp - *yp; /* x+y-y = x */
6       *xp = *xp - *yp; /* x+y-x = y */
7   }
```

如果调用这个过程时 xp 等于 yp，会有什么样的效果？

第二个妨碍优化的因素是函数调用。作为一个示例，考虑下面这两个过程：

```
1  int f(int);
2
3  int func1(x)
4  {
5       return f(x) + f(x) + f(x) + f(x);
6  }
7
8  int func2(x)
9  {
10      return 4*f(x);
11 }
```

最初看上去两个过程计算的都是相同的结果，但是 func2 只调用 f 一次，而 func1 调用 f 四次。以 func1 作为源时，会很想产生 func2 风格的代码。

不过，考虑下面 f 的代码：

```
1  int counter = 0;
2
3  int f(int x)
4  {
5      return counter++;
6  }
```

这个函数有个副作用——它修改了全局程序状态的一部分。改变调用它的次数会改变程序的行为。特别地，假设开始时全局变量 counter 都设置为 0，对 func1 的调用会返回 0+1+2+3=6，而对 func2 的调用会返回 4·0=0。

大多数编译器不会试图判断一个函数是否没有副作用，因此任意函数都可能是优化的候选者，例如 func2 中的做法。相反，编译器会假设最糟的情况，并保持所有的函数调用不变。

在各种编译器中，GNU 编译器 GCC 被认为是胜任的，但是就它的优化能力来说，并不是特别突出。它完成基本的优化，但是它不会对程序进行更加“有进取心的”编译器所做的那种激进变换。

因此，使用 GCC 的程序员必须花费更多的精力，以一种简化编译器生成高效代码的任务的方式来编写程序。

5.2 表示程序性能

我们需要一种方法来表示程序性能，它能指导我们改进代码。对许多程序都很有用的度量标准是*每元素的周期数*（cycles per element，CPE）。这种度量标准帮助我们在更详细的级别上理解迭代程序的循环性能。同时，这样的度量标准对执行重复计算的程序来说也是很适当的，例如处理图像中的像素，或是计算矩阵乘积中的元素。

处理器活动的顺序是由时钟控制的，时钟提供了某个频率的规律信号，要么用*兆赫兹*（MHz，即百万周每秒）来表示，要么用*千兆赫兹*（GHz，即吉周每秒）来表示。例如，一个系统有“1.4GHz”处理器，这表示处理器时钟运行频率为 1400 兆赫兹。每个时钟周期的时间是时钟频率的倒数，通常是用*纳秒*（nanosecond，十亿分之一秒）来表示的。一个 2GHz 的时钟其周期为 0.5 纳秒，而 500MHz 的时钟，周期为 2 纳秒。从程序员的角度来看，用时钟周期来表示度量标准要比用纳秒来表示有帮助得多。用时钟周期来表示，度量值不太依赖于被评估的处理器的模型，而这些度量值能帮助我们确切地理解机器是如何执行程序的。

许多过程含有在一组元素上迭代[1]的循环。例如，图 5.1 中的函数 vsum1 和 vsum2 计算的都是两个长度为 n 的向量之和。第一个函数每次迭代计算目标向量的一个元素。第二个函数使用称为*循环展开*（loop unrolling）的技术，每次迭代计算两个元素。这个版本只对 n 为偶数值有效。在本章后面，我们将更详细地介绍循环展开，包括如何使它对任意 n 的值都有效。

这样一个过程所需要的时间可以用一个常数加上一个与被处理元素个数成正比的因子来描述。例如，图 5.2 是这两个函数需要的每元素的周期数关于 n 值的取值范围图。使用*最小二乘方拟合*（least squares fit），我们发现，两个函数的运行时间（用时钟周期表示）分别近似于表达式 $80+4.0n$ 和 $83.5+3.5n$ 的线条。这两个表达式表明初始化过程、准备循环以及完成过程的开销为 80～84 个周期加上每个元素 3.5 或 4.0 周期的线性因子。对于较大的 n 的值（比如说，大于 50），运行时间就会主要由线性因子来决定。我们称这些项中的系数为*每元素的周期数*（简称 CPE）的有效数。注意，我们更愿意用每个元素的周期数而不是每次循环的周期数来度量，这是因为像循环展开这样的技术使得我们能够用较少的循环完成计算，而我们最终关心的是，对于给定的向量长度，程序运行的速度如何。我们将精力集中在减小我们计算的 CPE 上。根据这种度量标准，vsum2 的 CPE 为 3.5，优于 CPE 为 4.0 的 vsum1。

code/opt/vsum.c

```
1   void vsum1(int n)
2   {
3       int i;
4
5       for (i = 0; i < n; i++)
6           c[i] = a[i] + b[i];
7   }
8
```

1 本章中的迭代指执行一遍组成循环的语句块。——译者

```
9    /* Sum vector of n elements (n must be even) */
10   void vsum2(int n)
11   {
12       int i;
13
14       for (i = 0; i < n; i+=2) {
15           /* Compute two elements per iteration */
16           c[i] = a[i] + b[i];
17           c[i+1] = a[i+1] + b[i+1];
18       }
19   }
```

code/opt/vsum.c

图 5.1 向量求和函数

这是关于我们如何表示程序性能的示例。

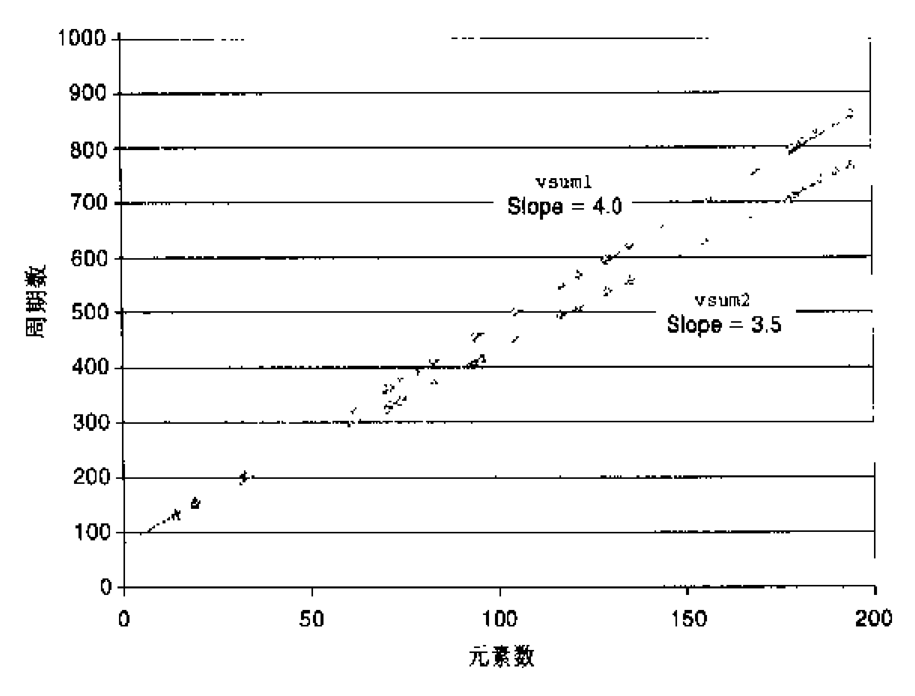


图 5.2 向量求和函数的性能

两条线的斜率表明每元素的周期数（CPE）。

旁注：什么是最小二乘方拟合？

对于一个数据点$(x_1, y_1),\cdots,(x_n, y_n)$的集合，我们常常试图画一条线，它能最接近于这些数据代表的趋势。使用最小二乘方拟合，我们寻找一条形如$y = mx + b$的线，使得下面这个误差度量最小：

$$E(m,b) = \sum_{i=1,n}(mx_i + b - y_i)^2$$

计算m和b的算法可以通过找到$E(m, b)$关于m和b的导数推导出来。

练习题 5.2

在本章后面，我们会采用一个函数，生成许多不同的变种，这些变种保持函数的行为，又具有

不同的性能特性。对于其中三个变种，我们发现运行时间（用时钟周期表示）可以用下面的函数近似地估计：

版本 1　60+35n

版本 2　136+4n

版本 3　157+1.25n

每个版本在 n 取什么值时是三个版本中最快的？记住，n 总是整数。

5.3 程序示例

为了说明一个抽象的程序是如何被系统地转换成更有效的代码的，考虑图 5.3 所示的简单向量数据结构。向量由两个存储器块表示。头部是一个声明如下的结构：

code/opt/vec.h

```
1    /* Create abstract data type for vector */
2    typedef struct {
3        int len;
4        data_t *data;
5    } vec_rec, *vec_ptr;
```

code/opt/vec.h

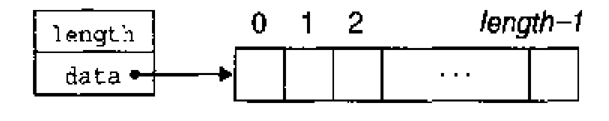


图 5.3　向量的抽象数据类型

向量由头信息加上指定长度的数组来表示。

这个声明用数据类型 data_t 作为基本元素的数据类型。在我们的评价中，我们度量我们的代码对于数据类型 int、float 和 double 的性能。为此，我们会分别为不同的类型声明编译和运行程序，就像下面这个例子中一样：

```
typedef int data_t;
```

除了头以外，我们还会分配一个 len 个 data_t 类型对象的数组，来存放实际的向量元素。

图 5.4 给出的是一些生成向量、访问向量元素以及确定向量长度的基本过程。一个值得注意的重要特性是 get_vec_element，向量访问程序会对每个向量引用进行边界检查。这段代码类似于许多其他语言（包括 Java）所使用的数组表示法。边界检查降低了程序出错的机率，但是正如我们看到的那样，它也明显影响了程序性能。

作为一个优化示例，考虑图 5.5 中所示的代码，它根据某种运算，将一个向量中所有的元素合并（combining）成一个值。通过使用编译时常数 IDENT 和 OPER 的不同定义，这段代码可以重编译成对数据执行不同的运算。特别地，使用声明：

```
#define IDENT 0
#define OPER  +
```

它对向量的元素求和。使用声明：

```
#define IDENT 1
```

```
#define OPER  *
```

它计算的是向量元素的乘积。

作为一个起点，下面是 combine1 的 CPE 度量值，它运行在 Intel PentiumIII 上，尝试了数据类型和合并运算的所有组合。在我们的度量值中，我们发现单、双精度浮点数据的时间基本上是相等的。因此，我们只给出对单精度浮点数据的度量值。

函数	页数	方法	整数		浮点数	
			+	*	+	*
Combine1	329	未优化的抽象的	42.06	41.86	41.44	160.00
Combine1	329	抽象的-O2	31.25	33.25	31.25	143.00

code/opt/vec.c

```
1   /* Create vector of specified length */
2   vec_ptr new_vec(int len)
3   {
4       /* allocate header structure */
5       vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
6       if (!result)
7           return NULL;  /* Couldn't allocate storage */
8       result->len = len;
9       /* Allocate array */
10      if (len > 0) {
11          data_t *data = (data_t *)calloc(len, sizeof(data_t));
12          if (!data) {
13              free((void *) result);
14              return NULL;  /* Couldn't allocate storage */
15          }
16          result->data = data;
17      }
18      else
19          result->data = NULL;
20      return result;
21  }
22
23  /*
24   * Retrieve vector element and store at dest.
25   * Return 0 (out of bounds) or 1 (successful)
26   */
27  int get_vec_element(vec_ptr v, int index, data_t *dest)
28  {
29      if (index < 0 || index >= v->len)
30          return 0;
31      *dest = v->data[index];
32      return 1;
33  }
34
35  /* Return length of vector */
```

```
36  int vec_length(vec_ptr v)
37  {
38      return v->len;
39  }
```

code/opt/vec.c

图 5.4 向量抽象数据类型的实现

在实际程序中，数据类型 data_t 被声明为 int、float 或 double。

code/opt/combine.c

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       int i;
5
6       *dest = IDENT;
7       for (i = 0; i < vec_length(v); i++) {
8           data_t val;
9           get_vec_element(v, i, &val);
10          *dest = *dest OPER val;
11      }
12  }
```

code/opt/combine.c

图 5.5 合并操作的初始实现

使用标识元素 IDENT 和合并运算 OPER 的不同声明，我们可以测量该函数对不同运算的性能。

默认地，编译器产生适合于用符号调试器一步一步调试的代码。因为目的是使目标代码尽可能类似于源代码中表明的计算，所以几乎没有进行什么优化。简单地将命令行开关设置为“-O2”，我们就能进行优化了。正如看到的那样，这显著地提高了程序性能。通常，养成进行这一级优化的习惯是很好的，除非编译程序就是为了要调试它。对于我们剩下的度量，我们都进行了这一级别的编译器优化。

还要注意，除了浮点数乘法以外，对于各种数据类型和不同运算的时间基本上都是同等的。浮点数乘法有很高的时钟周期数是由于我们基准程序数据中的异常。找出这样的异常是性能分析和优化的一个重要组成部分。我们会在 5.11.1 节中回过来讨论这个问题。我们会看到可以大幅度地提高它的性能。

5.4 消除循环的低效率

可以观察到，过程 combine1 调用函数 vec_length 作为 for 循环的测试条件，如图 5.5 所示。回想一下我们对循环的讨论，每次循环迭代时都必须对测试条件求值。另一方面，向量的长度并不会随着循环的进行而改变。因此，我们只需计算一次向量的长度，然后在我们的测试条件中使用这个值。

图 5.6 给出的是一个修改的版本，称为 combine2，它在开始时调用 vec_length，并将结果赋值

给局部变量 length。然后，在 for 循环的测试条件中使用这个局部变量。令人惊奇的是，这个小小的改动明显地影响了程序性能。如下表所示，通过这个简单的变换，我们为每个向量元素消除了大概10个时钟周期。

函数	页数	方法	整数		浮点数	
			+	*	+	*
combine1	329	抽象的-O2	31.25	33.25	31.25	143.00
combine2	330	移动 vec_length	22.61	21.25	21.15	135.00

图 5.6 改进循环测试的效率

通过把对 vec_length 的调用移出循环测试，我们不再需要每次迭代时都执行这个函数了。

code/opt/combine.c

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       int i;
5       int length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11           *dest = *dest OPER val;
12      }
13  }
```

code/opt/combine.c

这个优化是一类常见的、称为代码移动（code motion）的优化实例。这类优化包括识别出要执行多次（例如，在循环里）但是计算结果不会改变的计算，因而我们可以将计算移动到代码前面的、不会被多次求值的部分。在本例中，我们将对 vec_length 的调用从循环内部移动到循环的前面。

优化编译器会试着进行代码移动。不幸的是，就像前面讨论过的那样，对于会改变在哪里调用函数或调用多少次的变换，编译器通常会非常小心。它们不能可靠地发现一个函数是否会有副作用，因而它们会假设函数会有副作用。例如，如果 vec_length 有某种副作用，那么 combine1 和 combine2 可能就会有不同的行为。在这样的情况中，程序员必须帮助编译器显式地完成代码的移动。

作为 combine1 中看到的循环低效率的一个极端例子，考虑图 5.7 中所示的过程 lower1。这个过程是模仿几个学生的函数设计，他们的函数是作为一个网络编程项目的一部分提交的。这个过程的目的是将一个字符串中所有大写字母转换成小写字母。这个过程一步一步地检查字符串，将每个大写字符转换成小写字符。

code/opt/lower.c

```
1   /* Convert string to lower case: slow */
2   void lower1(char *s)
3   {
```

```
4       int i;
5
6       for (i = 0; i < strlen(s); i++)
7           if (s[i] >= 'A' && s[i] <= 'Z')
8               s[i] -= ('A' - 'a');
9   }
10
11  /* Convert string to lower case: faster */
12  void lower2(char *s)
13  {
14      int i;
15      int len = strlen(s);
16
17      for (i = 0; i < len; i++)
18          if (s[i] >= 'A' && s[i] <= 'Z')
19              s[i] -= ('A' - 'a');
20  }
21
22  /* Implementation of library function strlen */
23  /* Compute length of string */
24  size_t strlen(const char *s)
25  {
26      int length = 0;
27      while (*s != '\0') {
28          s++;
29          length++;
30      }
31      return length;
32  }
```

code/opt/lower.c

图 5.7 小写字母转换函数

两个过程的性能差别很大。

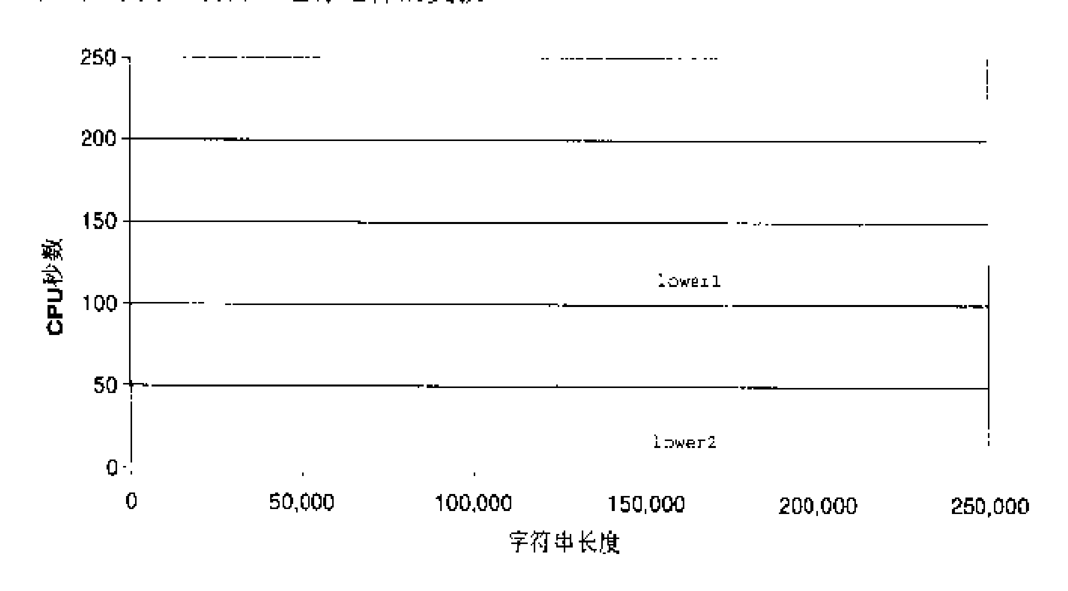
调用库过程 strlen 作为 lower1 的循环测试的一部分。图 5.7 中也给出了 strlen 的一个简单的版本。因为 C 中字符串是以 null 结尾的字符序列，strlen 必须一步一步地检查这个序列，直到遇到 null 字符。对于一个长度为 n 的字符串，strlen 所用的时间与 n 成正比。因为对 lower1 的 n 次迭代的每一次都会调用 strlen，所以 lower1 的整体运行时间是字符串长度的二次项。

如图 5.8 所示，这个过程对各种长度的字符串的实际测量值验证了上述分析。lower1 的运行时间曲线图随着字符串长度的增加上升得很陡峭。该图的下部展示了八个不同长度字符串的运行时间（与曲线图中所示的有所不同），每个长度都是 2 的幂数。可以观察到，对于 lower1 来说，字符串长度每增加一倍，运行时间都会变为原来的四倍。这很明显地表明复杂度是二次的。对于一个长度为 262144 的字符串，lower1 需要整整 3.1 分钟 CPU 时间。

除了我们把对 strlen 的调用移出了循环以外，图 5.7 中所示的 lower2 与 lower1 是一样的。这样

一来，性能有了显著改善。对于一个长度为 262144 的字符串，这个函数只需要 0.006 秒——比 lower1 快了 30000 多倍。字符串长度每增加一倍，运行时间也会增加一倍——很显然复杂度是线性的。对于较长的字符串，运行时间的改进会更大。

在理想的世界里，编译器会认出循环测试中对 strlen 的每次调用都会返回相同的结果，因此应该能够把这个调用移出循环。这需要非常成熟完善的分析，因为 strlen 会检查字符串的元素，而随着 lower1 的进行，这些值会改变。编译器需要探查，即使字符串中的字符发生了改变，但是没有字符会从非零变为零，或是反过来，从零变为非零。这样的分析远远超出了即使是最有野心的编译器的能力，所以程序员必须自己进行这样的变换。



函数	字符串长度					
	8 192	16 384	32 768	65 536	131 072	262 144
lower1	0.15	0.62	3.19	12.75	51.01	186.71
lower2	0.0002	0.0004	0.0008	0.0016	0.0031	0.0060

图 5.8 小写字母转换函数的性能比较

由于循环结构的效率比较低，原来的 lower1 的代码具有二次渐近（asymptotic）复杂性。修改过的 lower2 的代码有线性的复杂度。

这个示例说明了编程时一个常见的问题，一个看上去无足轻重的代码片断有隐藏的渐近低效率（asymptotic inefficiency）。人们可不希望一个小写字母转换函数成为程序性能的限制因素。通常，会在小数据集上测试和分析程序，对此，lower1 的性能是足够的。不过，当程序最终部署好以后，过程完全可能被应用到一个有 100 万个字符的串上，对此，lower1 从头至尾会需要 1 个小时的 CPU 时间。突然，这段无危险的代码变成了一个主要的性能瓶颈。相比较而言，lower2 会在 1 秒之内完成。大型编程项目中会出现这样的问题，这样的故事比比皆是。一个有经验的程序员工作的一部分就是避免引入这样的渐近低效率。

练习题 5.3

考虑下面的函数：

```
int min(int x, int y) { return x < y ? x : y; }
```

```
int max(int x, int y) { return x < y ? y : x; }
void incr(int *xp, int v) { *xp += v; }
int square(int x) { return x*x; }
```

下面三个代码片断调用这些函数：

```
A.      for (i = min(x, y); i < max(x, y); incr(&i, 1))
            t += square(i);
B.      for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
            t += square(i);
C.      int low = min(x, y);
        int high = max(x, y);

        for (i = low; i < high; incr(&i, 1))
            t += square(i);
```

假设x等于10，而y等于100。填写下表，指出在代码片断A~C中四个函数每个被调用的次数。

代码	min	max	incr	square
A.				
B.				
C.				

5.5 减少过程调用

像我们看到过的那样，过程调用会带来相当大的开销，而且妨碍大多数形式的程序优化。从combine2的代码（图5.6）中我们可以看出，每次循环迭代都会调用get_vec_element来获取下一个向量元素。这个过程开销特别大，因为它要进行边界检查。在处理任意的数组访问时，边界检查可能是个很有用的特性，但是对combine2的代码做简单的分析，表明所有的引用都是可以避免的。

作为替代，我们假设为我们的抽象数据类型增加一个函数get_vec_start。这个函数返回数组的起始地址，如图5.9所示。然后我们就能写出此图中combine3所示的过程，其中的循环里没有函数调用。它没有用函数调用来获取每个向量元素，而是直接访问数组。一个纯粹主义者可能会说这种变换严重地损害了程序的模块性。通常，向量抽象数据类型的使用者甚至不应该需要知道向量的内容是作为数组来存储的，而不是作为诸如链表之类的某种其他数据结构来存储的。比较实际的程序员会根据下面的实验结果，说明这种变换的优点：

函 数	页数	方 法	整 数		浮点数	
			+	*	+	*
combine2	330	移动vec_length	20.66	21.25	21.15	135.00
combine3	334	直接数据访问	6.00	9.00	8.00	117.00

code/opt/vec.c

```
1   data_t *get_vec_start(vec_ptr v)
2   {
```

```
3        return v->data;
4    }
```
code/opt/vec.c

code/opt/combine.c

```
1   /* Direct access to vector data */
2   void combine3(vec_ptr v, data_t *dest)
3   {
4       int i;
5       int length = vec_length(v);
6       data_t *data = get_vec_start(v);
7
8       *dest = IDENT;
9       for (i = 0; i < length; i++) {
10          *dest = *dest OPER data[i];
11      }
12  }
```
code/opt/combine.c

图 5.9 消除循环中的函数调用

得到的代码运行速度比较快，是以损害一些程序的模块性为代价的。

改进最高可以达到 3.5X。对于性能至关重要的程序来说，为了速度，经常必须要损害一些模块性和抽象性。如果以后需要修改代码，添加一些对所采用的变换进行记录的文档是很明智的。

旁注：表示相对性能

表示性能改进最好的方法就是形如 Told/Tnew 的比率，这里 Told 是原始版本所需的时间，而 Tnew 是修改过的版本所需的时间。如果发生了实际的改进，它应该是一个大于 1.0 的数字。我们用后缀"X"来表示这样一种比率，因子"3.5X"读作"3.5 倍"。

更加传统的表示相对变化的方法是百分比，在变化很小时，还是很有效的，但是它的定义十分含糊。它应该是 100·(Told−Tnew)/Tnew，还是 100·(Told−Tnew)/Told，或是别的什么呢？此外，对于较大的变化，它就不那么有帮助了。说"性能提高了 250%"比简单的说性能改进因子为 3.5 要更难以理解一些。

5.6 消除不必要的存储器引用

combine3 的代码将合并操作计算的值累积在指针 dest 指定的位置。通过检查被编译的循环产生的汇编代码，整数作为数据类型，乘法作为合并操作，可以看出这个属性。在这段代码中，寄存器 %ecx 指向 data，%edx 包含 i 的值，而%edi 指向 dest。

```
    combine3:  type=INT, OPER = *
    dest in %edi, data in %ecx, i in %edx, length in %esi
1   .L18:                              loop:
2     movl (%edi),%eax                   Read *dest
3     imull (%ecx,%edx,4),%eax           Multiply by data[i]
4     movl %eax,(%edi)                   Write *dest
5     incl %edx                          i++
```

```
6        cmpl %esi,%edx                    Compare i:length
7        jl .L18                           If <,goto loop
```

指令 2 读取存放在 dest 中的值，指令 4 写回这个位置。这看上去是种浪费，因为正常情况下，下一次迭代时指令 2 读取的值会是刚刚写回的那个值。

这就导致了图 5.10 中 combine4 所示的优化，在这里，我们引入了一个临时变量 x，它用在循环中存放计算出来的值。只有在循环完成之后结果才存放在*dest 中。正如下面的汇编代码所示，编译器现在可以用寄存器%eax 保存累积值。与 combine3 的循环相比，我们将每次迭代的存储器操作从两次读和一次写减少到只需要一次读。寄存器%ecx 和%edx 的使用和前面一样，但是不再需要引用*dest。

```
    combine4: type=INT, OPER = *
    data in %eax, x in %ecx, i in %edx, length in %esi
1     .L24:                                loop:
2       imull (%eax,%edx,4),%ecx             Multiply x by data[i]
3       incl %edx                            i++
4       cmpl %esi,%edx                       Compare i:length
5       jl .L24                              If <, goto loop
```

code/opt/combine.c

```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       int i;
5       int length = vec_length(v);
6       data_t *data = get_vec_start(v);
7       data_t x = IDENT;
8
9       *dest = IDENT;
10      for (i = 0; i < length; i++) {
11          x = x OPER data[i];
12      }
13      *dest = x;
14  }
```

code/opt/combine.c

图 5.10 在临时变量中存放结果

这使得每次循环迭代中不再需要读和写中间值。

我们看到程序性能有了显著的改善，如下表所示：

函 数	页数	方 法	整 数		浮点数	
			+	*	+	*
combine3	334	直接数据访问	6.00	9.00	8.00	117.00
combine4	335	累积在临时变量中	2.00	4.00	3.00	5.00

下降最快的是浮点数乘法的时间。它的时间变得和其他数据类型和操作的组合所用时间可比较了。我们会在 5.11.1 小节中检查这种迅速下降的原因。

可能又有人会认为编译器应该能够自动将图5.9中所示的combine3的代码转换为在寄存器中存放的那个值，就像图5.10中所示的combine4的代码所做的那样。

然而实际上，由于存储器别名的使用，两个函数可能会有不同的行为。例如，考虑整数数据，运算为乘法，标识元素为1的情况。v是一个由三个元素[2, 3, 5]组成的向量，考虑下面两个函数调用：

```
combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);
```

也就是，我们在向量最后一个元素和存放结果的目标之间创建一个别名。那么，两个函数的执行如下：

函数	初始值	循环之前	i = 0	i = 1	i = 2	最后
combine3	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
combine4	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]

正如前面讲到过的，combine3将它的结果存放在目标位置中，在本例中，目标位置就是向量的最后一个元素。因此，这个值首先被设置为1，然后设为2 · 1 = 2，然后设为3 · 2 = 6。最后一次迭代中，这个值会乘以它自己，得到最后结果36。对于combine4的情况来说，直到最后向量都保持不变，结束之前，最后一个元素会被设置为计算出来的值1 · 2 · 3 · 5 = 30。

当然，我们说明combine3和combine4之间差别的例子是人为设计的。有人会说combine4的行为更加符合函数描述的意图。不幸的是，优化编译器不能判断函数会在什么情况下被调用，以及程序员的本意可能是什么。取而代之的是，在编译combine3时，编译器有责任保持它的功能，即使这意味着生成低效率的代码。

5.7 理解现代处理器

到目前为止，我们运用的优化都不依赖于目标机器的任何特性。这些优化只是简单地降低了过程调用的开销，以及消除了一些重大的“妨碍优化的因素”，这些因素会给优化编译器造成困难。随着我们试图进一步提高性能，我们必须开始考虑这样的优化，它们更多地利用处理器执行指令的方式和某些处理器的能力。要想获得最大的性能，需要仔细地分析程序，同时代码的生成也要针对目标处理器进行调整。尽管如此，我们还是能够运用一些基本的优化，在很大一类处理器上产生整体的性能提高。我们在这里公布的详细性能结果，对其他机器不一定也有同样的效果，但是操作和优化的通用原则对范围众多的机器都适用。

为了理解改进性能的方法，我们需要一个关于现代处理器是如何工作的简单操作模型。由于大量的晶体管可以被集成到一块芯片上，现代微处理器采用了复杂的硬件，试图使程序性能最大化。一个后果就是处理器的实际操作与观察汇编语言程序得到的概念大相径庭。在汇编代码级，看上去似乎是一次执行一条指令，每条指令都包括从寄存器或存储器取值，执行一个操作，并把结果存回到一个寄存器或存储器位置。在实际的处理器中，是同时对多条指令求值的。在某些设计中，可以有80或更多条指令在处理中。采用一些精细的机制来确保这种并行执行的行为，能正好获得机器级程序要求的顺序语义模型的效果。

5.7.1 整体操作

图 5.11 给出了现代微处理器的一个非常简单化的示意图。我们假设的处理器设计是基于 Intel “P6”微体系结构的[30]，这种微体系结构是 Intel PentiumPro、Pentium II 和 Pentium III 处理器的基础。较新的 Pentium 4 的微体系结构有所不同，不过它的整体结构与我们在这里讲述的很类似。P6 微体系结构是自 20 世纪 90 年代后期以来许多厂商生产的高端处理器的典型。在工业界称为超标量（superscalar），意思是它可以在每个时钟周期执行多个操作，而且是乱序的（out-of-order），意思就是指令执行的顺序不一定要与它们在汇编程序中的顺序一致。整个设计有两个主要部分：ICU（Instruction Control Unit，指令控制单元）和 EU（Execution Unit，执行单元）。前者负责从存储器中读出指令序列，并根据这些指令序列生成一组针对程序数据的基本操作；而后者执行这些操作。

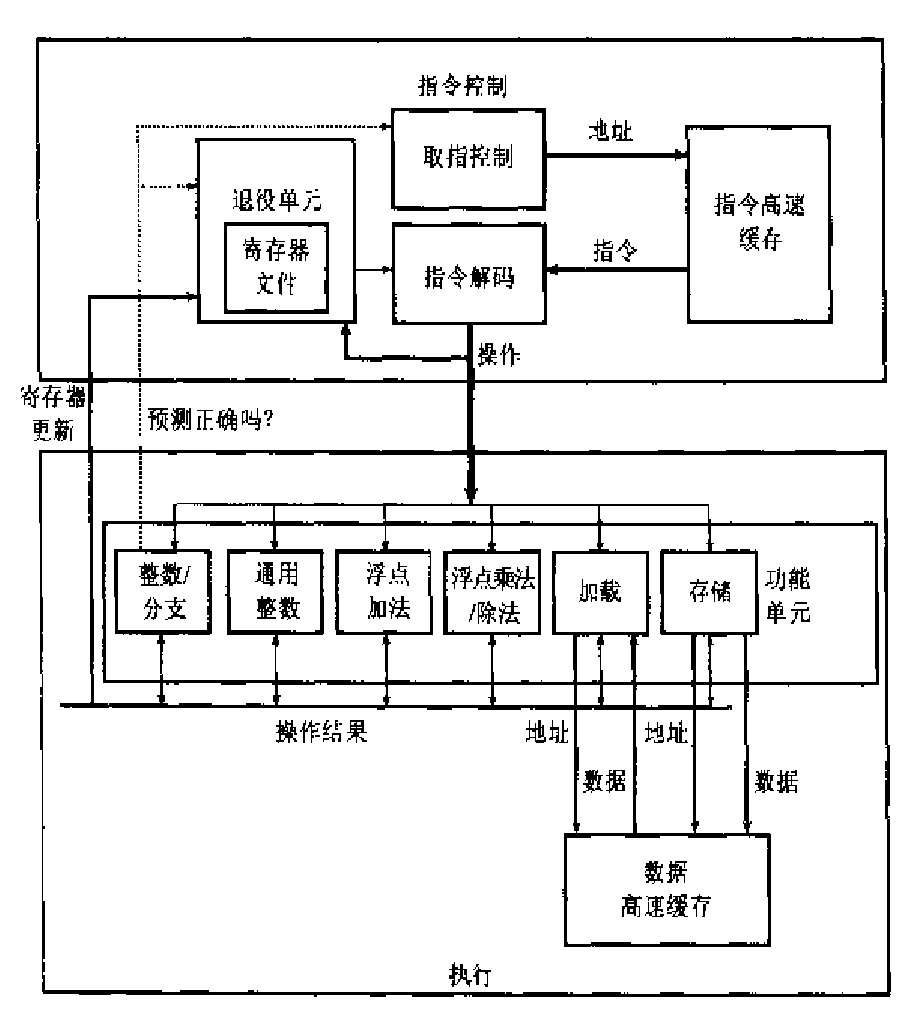


图 5.11 一个现代处理器的框图

指令控制单元负责从存储器中读出指令，并产生一系列基本操作。然后执行单元完成这些操作，以及指出分支预测是否正确。

ICU 从指令高速缓存（instruction cache）中读取指令，指令高速缓存是一个特殊的高速缓存存储器，它包含最近访问的指令。通常，ICU 会在当前正在执行的指令很早之前取指，所以它有足够的时间对指令解码，并把操作发送到 EU。不过，有一个问题，那就是当程序遇到分支[1]时，

1 我们用术语“分支”专指条件转移指令。对处理器来说，其他可能将控制传送到多个目的地址的指令，例如过程返回和间接跳转，处理起来的困难程度是类似的。

程序有两个可能的前进方向。一种可能会选择分支，控制被传递到分支目标；另一种可能是，不选择分支，控制被传递到指令序列的下一条指令。现代处理器采用了一种称为分支预测（branch prediction）的技术，在这种技术中处理器会预测是否选择分支，同时还预测分支的目标地址。使用一种称为投机执行（speculative execution）的技术，处理器会开始取出它预测的分支处的指令并对指令解码，甚至于在它确定分支预测是否正确之前就开始执行这些操作。如果过后它确定分支预测错误，它会将状态重新设置到分支点的状态，并开始取出和执行另一个方向上的指令。一种更加异乎寻常的技术是开始取出和执行两个可能方向上的指令，随后再抛弃掉不正确方向上的结果。时至今日，都不认为这种方法的成本效率是值得的。标号为取指控制的块包括分支预测，以完成确定取哪条指令的任务。

指令解码逻辑接收实际的程序指令，并将它们转换成一组基本操作。每个操作都完成某个简单的计算任务，例如两个数相加，从存储器中读数据，或是向存储器写数据。对于具有复杂指令的机器，比如说像 IA32 处理器，可能将一条指令解码成可变数量的操作。每个处理器设计的详细情况都有所不同，但是我们试着描述一种典型的实现。在这种机器上，解码下面这个指令

```
addl %eax, %edx
```

产生一个加法操作，而解码下面这个指令

```
addl %eax, 4(%edx)
```

产生三个操作——一个操作从存储器中加载一个值到处理器中，一个操作将加载进来的值加上寄存器%eax 中的值，而一个操作将结果存回到存储器。这种解码逻辑分解指令的操作，实现了在一组专门的硬件单元之间的任务分割。然后，这些单元可以并行地执行乘法指令的各个部分。对于具有简单指令的机器，操作更紧密地对应于原始的指令。

EU 接收来自指令读取单元的操作。通常，它会每个时钟周期接收若干个操作。这些操作会被分派到一组功能单元中，它们会执行实际的操作。这些功能单元是专门用来处理特定类型的操作。我们的图（指图 5.11）说明了一组典型的功能单元。它沿用的是最近的 Intel 处理器的风格。图中的单元如下：

整数/分支：执行简单的整数操作（加法、测试、比较、逻辑）。还处理分支，就像下面会讨论的那样。

通用整数：可以处理所有的整数操作，包括乘法和除法。

浮点加法：处理简单的浮点操作（加法、格式转换）。

浮点乘法/除法：处理浮点乘法和除法。更复杂的浮点指令，例如超越函数（transcendental function），会被转换成操作的序列。

加载：处理从存储器读数据到处理器的操作。这个功能单元有一个加法器来执行地址计算。

存储：处理从处理器到存储器的写操作。这个功能单元有一个加法器来执行地址计算。

如图中所示，加载和存储单元通过数据高速缓存访问存储器，这是一个高速存储器，包含最近访问的数据值。

使用投机执行技术，对操作求值，但是最终结果不会存放在程序寄存器或数据存储器中，直到处理器能确定应该实际执行这些指令。分支操作被送到 EU，不是确定分支该往哪里去，而是确定分支预测是否正确。如果预测错误，EU 会丢弃分支点之后计算出来的结果。它还会发信号给分支

单元，说预测是错误的，并指出正确的分支目的。在这种情况中，分支单元开始读取新位置的指令。这样的预测错误会导致很大的性能开销。在可以取出新指令、解码和发送到执行单元之前，要花费一点时间。我们会在 5.12 节中进一步研究这个问题。

在 ICU 中，*退役单元*（Retirement Unit）记录正在进行的处理，并确保它遵守机器级程序的顺序语义。我们的图中展示了一个*寄存器文件*，它包含整数和浮点数寄存器，是退役单元的一部分，因为退役单元控制这些寄存器的更新。指令解码时，关于指令的信息被放置在一个先进先出的队列中。这个信息会一直保持在队列中，直到两个结果中的一个发生。首先，一旦指令的操作完成了，而所有导致这条指令的分支点也都被确认为预测正确，那么这条指令就可以*退役*了，所有对程序寄存器的更新都可以被实际执行了。另一方面，如果导致该指令的某个分支点预测错误，这条指令会被清空，丢弃所有计算出来的值。通过这种方法，错误的预测就不会改变程序状态了。

正如我们已经描述的那样，任何对程序状态的更新都只会在指令退役时才会发生，只有在处理器能够确信导致这条指令的所有分支都预测正确了，才能这样做。为了加速一条指令到另一条指令的结果的传送，许多此类信息是在执行单元之间交换的，即图中的"操作结果"。如图中的箭头所示，执行单元可以直接将结果发送给彼此。

最常见的控制操作数在执行单元间传送的机制称为*寄存器重命名*（register renaming）。当一条更新寄存器 r 的指令解码时，产生标记 t（tag t），得到一个指向该操作结果的惟一的标识符。条目 (r,t) 被加入到一张表中，该表维护着每个程序寄存器与会更新该寄存器的操作的标记之间的关联。当随后以寄存器 r 作为操作数的指令解码时，发送到执行单元的操作会包含 t 作为操作数源的值。当某个执行单元完成第一个操作时，会生成一个结果 (v,t)，指明标记为 t 的操作产生值 v。此时，所有等待 t 作为源的操作都能使用 v 作为源值了。通过这种机制，值可以直接从一个操作传递到另一个操作，而不是写到寄存器文件再读出来。重命名表只包含关于有未进行写操作的寄存器条目。当一条已解码的指令需要寄存器 r，而又没有标记与这个寄存器相关联，这个操作数可以直接从寄存器文件中获得。有了寄存器重命名，即使只有在处理器确定了分支结果之后才能更新寄存器，也可以预测着执行操作的整个序列。

旁注：乱序处理的历史

乱序处理最早是在 1964 年 Control Data 公司的 6600 处理器中实现的。指令是由十个不同的功能单元处理的，每个单元都能独立地操作。在那个时候，这种时钟频率为 10Mhz 的机器被认为是科学计算最好的机器。

在 1966 年，IBM 首先是在 IBM 360/91 上实现了乱序处理，但只是用来执行浮点指令。在大约 25 年的时间里，乱序处理都被认为是一项异乎寻常的技术，只在追求尽可能高性能的机器中使用，直到 1990 年 IBM 在 RS/6000 系列工作站中重新引入了这项技术。这种设计成为了 IBM/Motorola PowerPC 系列的基础，典型代表是 1993 年引入的 601，它成为第一个使用乱序处理的单芯片微处理器。

5.7.2 功能单元的性能

图 5.12 提供了 Intel Pentium III 的一些基本操作的性能，其他处理器也具有这样的计时特征。每个操作都是由两个周期计数值来刻画的：一个是*执行时间*（latency），它指明功能单元完成操作所需要的总周期数；另一个是*发射时间*（issue time），它指明连续的、独立操作之间的周期数。执行时

间的范围从基本整数操作的一个周期，到加载、存储、整数乘法和更常见的浮点操作的几个周期，到除法和其他复杂操作的许多个周期。

正如图 5.12 中第三栏所示，处理器的几个功能单元被流水线化了，这意味着在前一个操作完成之前，它们就可以开始一个新的操作。发射时间指明一个单元的连续操作之间的周期数。在一个流水线化的单元中，发射时间比执行时间短。流水线化的功能单元是作为一系列阶段来实现的，每个阶段完成操作的一部分。例如，一个典型的浮点加法器包含三个阶段：一个阶段处理指数值，一个将小数相加，而一个四舍五入计算最后的结果。操作可以连续地通过各个阶段，而不是等待一个操作完成后再开始下一个。只有当要执行的操作是连续的、逻辑上独立的，才能运用这种功能。正如表明的那样，大多数单元能够每个时钟周期开始一个新的操作。仅有的例外是浮点乘法器和两个除法器，浮点乘法器要求连续的操作之间至少要有两个周期，而两个除法器根本就没有流水线化。

操　作	执行时间	发射时间
整数加法	1	1
整数乘法	4	1
整数除法	36	36
浮点加法	3	1
浮点乘法	5	2
浮点除法	38	38
加载（高速缓存命中）	3	1
存储（高速缓存命中）	3	1

图 5.12 Pentium III 算术操作的性能

执行时间代表一条操作的总周期数。发射时间表示连续的、独立的操作之间的周期数（来自于 Intel 的文献）。

电路设计者可以创建具有一系列性能特性的功能单元。创建一个执行时间短或发射时间短的单元需要较多的硬件，特别是对于像乘法和浮点操作这样比较复杂的功能。因为微处理器芯片上，对于这些单元，只有有限的空间，所以 CPU 设计者必须小心地平衡功能单元的数量和它们各自的性能，以获得最优的整体性能。设计者们评估许多不同的基准程序，将大多数资源用于最关键的操作。如图 5.12 表明的那样，在 Pentium III 的设计中，整数乘法、浮点乘法和加法被认为是重要的操作，即使需要大量硬件以获得低执行时间和较高的流水线化程度。另一方面，除法相对不太常用，而且难以实现低执行时间或发射时间，因此这些操作相对而言比较慢。

5.7.3 更近地观察处理器操作

作为分析在现代处理器上执行的机器级程序的性能，我们提出了一种更详细的文本表示法来描述指令解码器产生的操作，还有一种图形化的表示法来显示功能单元对操作的处理。这两种表示法都不能准确地表示具体的、现实的处理器的实现，它们是简单的方法，帮助理解处理器在执行程序时能够如何利用并行性和分支预测。

将指令翻译成操作

我们通过 combine4（图 5.10）来说明我们的表示法，它是到目前为止我们最快代码的示例。我们只关注循环执行的操作，因为对很大的向量来说，这是性能的决定性因素。我们考虑整数数据以及以乘法和加法作为合并操作的情况。使用乘法的循环的编译代码由四条指令组成。在这个代码中，寄存器%eax 保存指针 data，%edx 保存 i，%ecx 保存 x，而%esi 保存 length：

```
        combine4: type=INT, OPER = *
        data in %eax, x in %ecx, i in %edx, length in %esi
1     .L24:                                    loop:
2       imull (%eax,%edx,4),%ecx                 Multiply x by data[i]
3       incl %edx                                i++
4       cmpl %esi,%edx                           Compare i:length
5       jl .L24                                  If <, goto loop
```

每次处理器执行这个循环时，指令解码器将这四条指令翻译成执行单元的一个操作序列。第一次迭代时，i 等于 0，我们假定的机器会发射下面的操作序列：

汇编指令	执行单元操作
.L24:	
imull (%eax,%edx,4),%ecx	load (%eax, %edx.0, 4) → t.1
	imull t.1, %ecx.0 → %ecx.1
incl %edx	incl %edx.0 → %edx.1
cmpl %esi,%edx	cmpl %esi, %edx.1 → cc.1
jl .L24	jl-taken cc.1

在我们的指令翻译中，我们将乘法指令的存储器引用转换成一条显式的 load 指令，它将数据从存储器读到处理器。我们还给每次迭代都变化的值分配操作数标号（operand label)。这些标号是寄存器重命名生成的标记的风格化版本。因此，循环开始处，寄存器%ecx 中的值由标号%ecx.0 标识，在更新后，由%ecx.1 标识。这次迭代与下次迭代不变化的寄存器值可以在解码时直接从寄存器文件中获得。我们还引入了标号 t.1，来表示 load 操作读取的、传送到 imull 操作的值，而我们显式地给出了操作的目的地。因此，一对操作

```
load (%eax, %edx.0, 4)  → t.1
imull t.1, %ecx.0       → %ecx.1
```

表明，处理器首先执行一条 load 操作，用%eax 的值（这个值在循环中不会改变）和循环开始时存放在%edx 中的值来计算地址。这会产生一个临时值，标号为 t.1。然后，乘法操作获取这个值和循环开始时%ecx 的值，产生一个%ecx 的新值。正如这个例子说明的那样，标记可以与并不会写到寄存器文件中的中间值相关联。

操作

```
incl %edx.0 → %edx.1
```

指明，增量操作对循环开始时%edx 的值加 1，产生这个寄存器的新值。

操作

```
cmpl %esi, %edx.1 → cc.1
```

指明，比较操作（由两个整数单元中的一个执行）比较%esi 中的值（这个值在循环中不会改变）和新计算出来的%edx 的值。然后，它会设置标号 cc.1 标识的条件码。正如这个例子说明的那样，处理器可以用重命名来记录对条件码寄存器的改变。

最后，预测跳转指令会选择分支。跳转指令

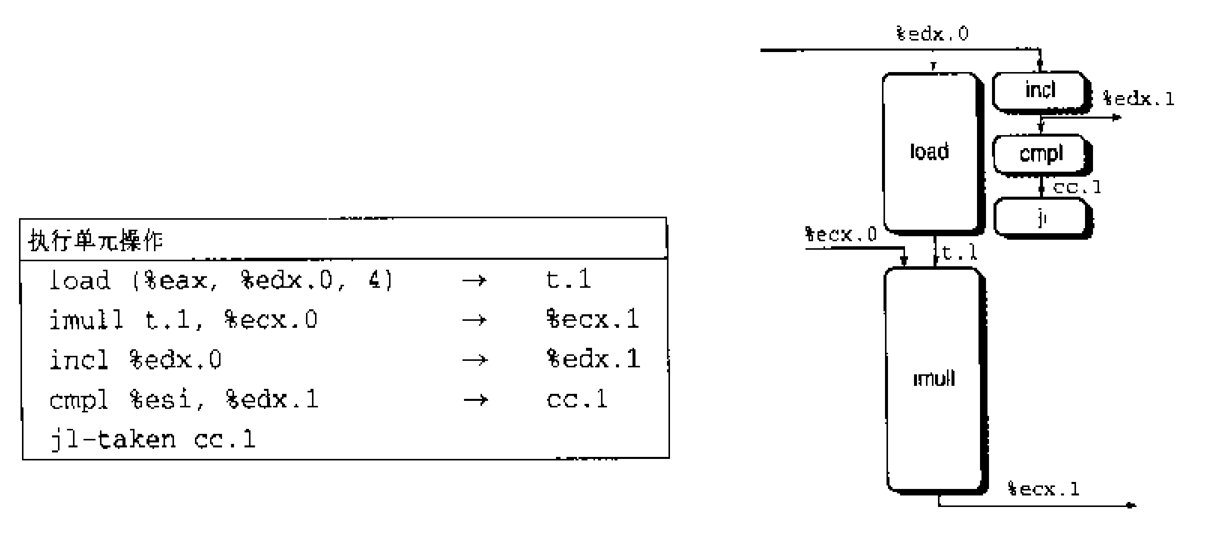
```
jl-taken cc.1
```

检查新计算出来的条件码的值（cc.1）是否表明这是个正确的选择。如果不是，那么它会发信号给ICU，告诉它在 jl 后面的指令处开始取指令。为了简化表示法，我们省略了所有关于可能的跳转目的地的信息。实际上，处理器必须记录未被预测方向的目的地，这样一来，在预测错误时，它可以从那里开始取指。

如这个示例翻译表明的那样，我们的操作在许多方面模仿了汇编语言指令的结构，除了它们是用标识寄存器不同实例的标号来引用它们的源和目的操作的。在实际的硬件中，寄存器重命名动态地给标记赋值，使之指向这些不同的值。标记是位模式而不是像“%edx.1”这样的符号名字，但是它们提供的用途是一样的。

执行单元的操作处理

图 5.13 以两种形式展示了操作：一种是指令解码器生成的形式，另一种是用计算图（computation graph）来表示的，在这种图中，操作是用圆角方框表示的，而箭头表明操作之间的数据传递。我们只为一次迭代与下一次迭代之间改变了的操作数而显示箭头，因为只有这些值才在功能单元之间进行传递。



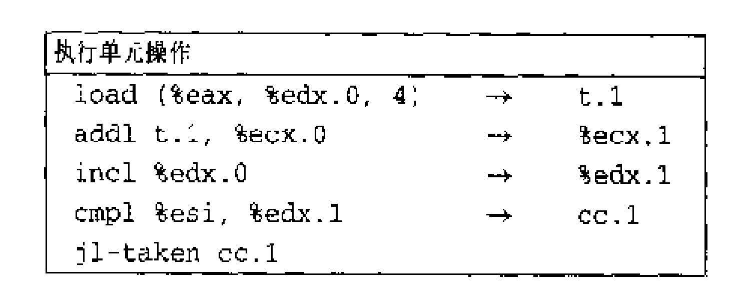
执行单元操作		
`load (%eax, %edx.0, 4)`	→	`t.1`
`imull t.1, %ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

图 5.13 整数乘法的 combine4 的内循环第一次迭代的乘法操作

存储器读被显式地转换成了加载。寄存器名字是用实例号码（instance number）标记的。

每个操作符方框的高度表明这个操作需要多少个周期，也就是这一种功能的执行时间。在此，整数乘法 imull 需要四个周期，加载需要三个周期，而其他操作需要一个周期。在展示一个循环的计时中，我们将块竖直地放置，来表示操作执行的时间，向下的方向表示时间的增长。我们可以看到，循环的五个操作形成了两个并行的链，表明两个计算序列必须顺序地执行。左边的链处理数据，首先从存储器中读一个数组元素，然后用它乘以累积的乘积。右边的链处理循环索引 i，首先对它加 1，然后拿它与 length 做比较。跳转操作检查这个比较的结果，以确定分支预测是正确的。注意，从跳转操作方框中没有向外的箭头。如果分支预测正确，不需要任何处理。如果分支预测错误，那么分支功能单元会发信号给指令取出控制单元，而这个单元会采取改正的行动。无论是两种情况中的哪一种，其他的操作都不依赖于跳转操作的结果。

图 5.14 给出了同样的到操作的翻译，只不过合并操作是整数加法。如图形描述所示，所有的操作，除了加载以外，现在都只需要一个周期。



执行单元操作		
load (%eax, %edx.0, 4)	→	t.1
addl t.1, %ecx.0	→	%ecx.1
incl %edx.0	→	%edx.1
cmpl %esi, %edx.1	→	cc.1
jl-taken cc.1		

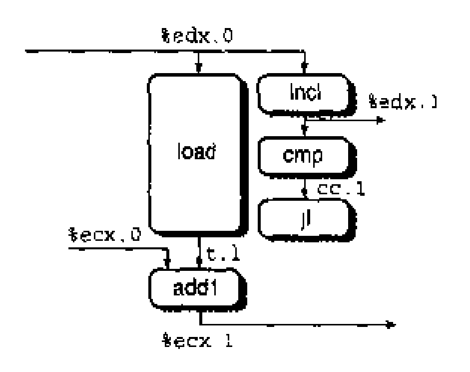


图 5.14　整数加法的 combine4 里面循环第一次迭代的操作

与乘法相比，惟一的变化是加法操作只需要一个周期。

有无限资源的操作调度

为了看看处理器将如何执行一系列的反复，首先设想一个处理器，它有无限多个功能单元和完美的分支预测。只要一个操作的数据操作数可用，该操作就能够开始执行了。这样一个处理器的性能只受下列因素的限制：功能单元的执行时间和吞吐量，以及程序中的数据相关性。图 5.15 给出的是在这样一个机器上整数乘法的 combine4 中循环头三次迭代的计算图。对每次迭代，都有一组五个操作，形式与图 5.13 中所示的一样，而操作数标号有适当的变化。从一次迭代的操作数到另一次迭代的操作数的箭头表明了各个迭代之间的数据相关。

根据没有朝上的箭头这一限制条件，每个操作都竖直地放在尽可能高的位置，因为朝上的箭头表明信息流向过去的时间。因此，只要前一次迭代的 incl 操作产生了循环索引（loop index）的更新值，下一次迭代的 load 操作就能够开始了。

这个计算图展示了执行单元对操作的并行执行。每个周期中，图上一条水平线上的所有操作是并行执行的。这个图还展示了乱序和投机执行。例如，一次迭代中的 incl 操作在前一次迭代的 jl 指令开始之前就执行了。我们还能看到流水线化的效果。每次迭代从头至尾至少需要七个周期，但是随后的迭代每四个周期就能完成。因此，有效处理频率是每四周期一次迭代，CPE 为 4.0。

整数乘法四个周期的执行时间限制了处理器对这个程序的性能。每个 imull 操作必须等待直到前一个操作完成，因为在开始之前，它需要这次乘法的结果。在我们的图中，乘法操作在周期 4、8 和 12 上开始。在随后的迭代中，每四个周期开始一条新的乘法。

图 5.16 展示了在一个有无限多个功能单元的机器上，整数加法的 combine4 的头四次迭代。如果合并操作只需要一个周期，程序的 CPE 就能达到 1.0。我们看到随着循环的进行，执行单元就能每个时钟周期执行七个操作的一部分了。例如，在周期 4 中，我们可以看到机器在执行迭代 1 的 addl，迭代 2、3 和 4 的 load 操作的不同部分，迭代 2 的 jl，迭代 3 的 cmpl 以及迭代 4 的 incl。

资源约束下的操作调度

当然，一个真实的处理器只有固定数目的功能单元。和我们前面的例子不同，在那些例子中，性能只受数据相关性和功能单元的执行时间的限制，现在性能还受资源约束的限制。特别地，我们的处理器只有两个单元能执行整数和分支操作。相反，在图 5.15 中，周期 3 中有三个此类操作在并行执行，而周期 4 中有四个在并行执行。

图 5.17 展示了在一个有资源约束的处理器上，整数乘法的 combine4 的操作调度。我们假设通

用的整数单元和分支/整数单元都能够每个周期开始一个新操作。可能有多于两个的整数或分支操作并行地执行，就像周期6中所示的那样，因为此时 imull 操作处在它的第三个周期。

因为资源受约束，我们的处理器必须要有**调度策略**，在有多个选择时，它要确定应该执行哪个操作。例如，图5.15中图表的周期3中，我们展示了三个正在被执行的整数操作：迭代1的 jl、迭代2的 cmpl 和迭代3的 incl。对于图5.17来说，我们必须推迟这些操作中的一个。我们通过记录操作的**程序顺序**（program order）来做到这一点，程序顺序也就是如果我们按照严格的顺序来执行机器级程序，操作执行的顺序。那么我们会根据操作的程序顺序赋给它们优先级。在此例中，我们会推迟 incl 操作，因为迭代3的任何操作在程序顺序中都在迭代1和2的操作之后。类似地，在周期4中，我们会使迭代1的 imull 操作和迭代2的 jl 操作的优先级高于迭代3的 incl 操作的优先级。

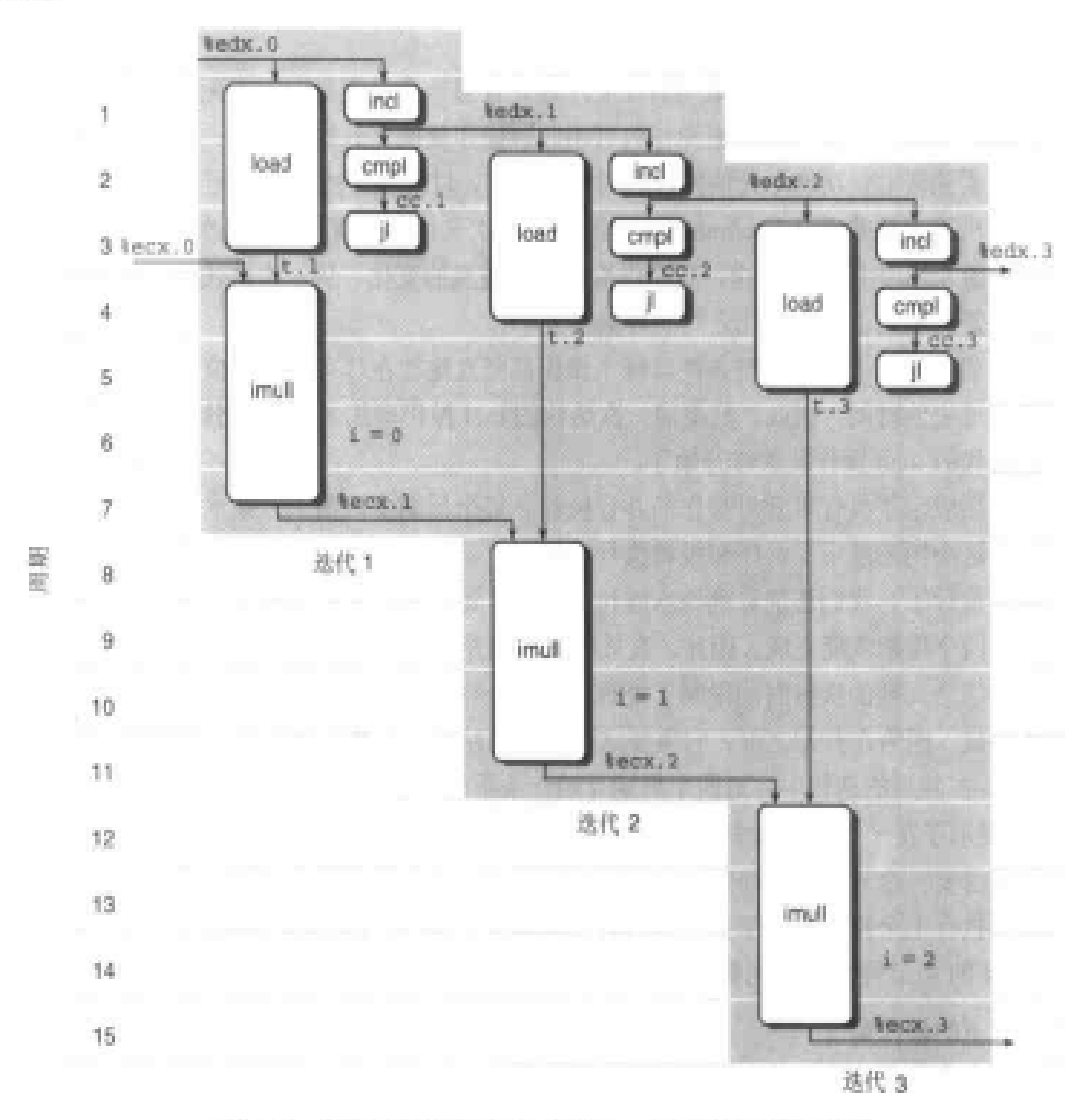


图5.15 执行单元数量无限的情况下，整数乘法操作的调度

乘法器的4周期执行时间是限制性能的资源。

对于这个例子来说，功能单元数量有限并没有使我们的程序变慢。性能仍然是被整数乘法的四周期执行时间限制的。

对于整数加法的情况，资源约束明显地限制了程序性能。每次迭代要有四个整数或分支操作，而只有两个功能单元能完成这些操作。因此，我们不能期望能保持比每次迭代两个周期更好的处理频率了。在创建整数加法的 combine4 的多次迭代的图表时，出现了一种很有趣的模式。图 5.18 展示了迭代 4~8 的操作的调度。我们选择这个范围内的迭代，是因为它展示了操作时间的规则模式（regular pattern）。在迭代 4~8 中，所有操作出现的时间都是相同的，除了迭代 8 中的操作的发生晚了八个周期。随着迭代的进行，迭代 4~7 所示的模式会不断重复。因此，我们每八个周期完成四次迭代，得到最优的 CPE 2.0。

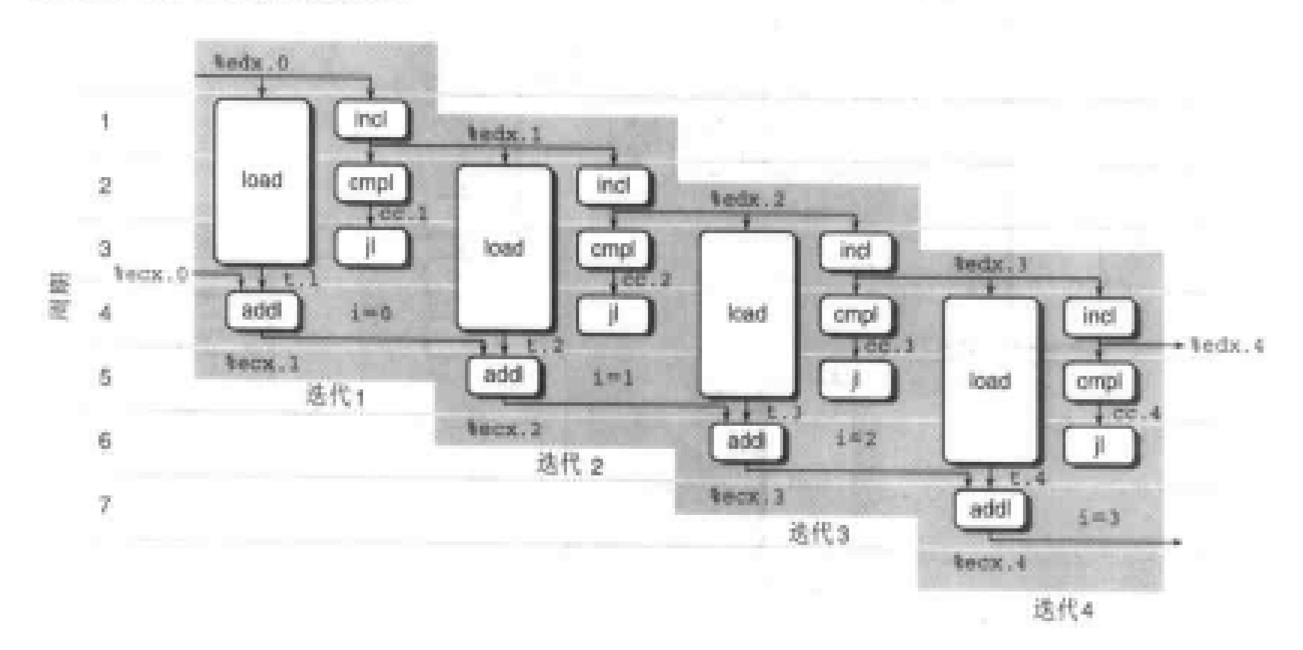


图 5.16　在不受限制的资源约束的情况下，整数加法操作的调度

如果资源不受限制，处理器的 CPE 能达到 1.0。

combine4 性能小结

现在我们来考虑一下 combine4 对四种组合数据类型和合并操作的测量性能：

函　数	页数	方　法	整　数		浮点数	
			+	*	+	*
combine4	335	累积在临时变量中	2.00	4.00	3.00	5.00

除了整数加法例外以外，这些周期时间基本上都与合并操作的执行时间相符，如图 5.12 所示。在此，我们的转换将 CPE 值降低到合并操作的时间成为限制因素。

对于整数加法的情况，我们看到，有限数量的针对分支和整数操作的功能单元限制了能达到的性能。每次迭代有四个这类操作，而只有两个功能单元，我们不能指望程序能运行得比每次迭代 2 个周期更快了。

通常，处理器性能是受三类约束限制的。第一，程序中的数据相关性迫使一些操作延迟直到它

们的操作数被计算出来。因为功能单元有一个或多个周期的执行时间，这就设置了一个给定的操作序列执行周期数的下界。第二，资源约束限制了在任意给定时刻能够执行多少个操作。我们看到，功能单元的有限数量就是这样一种资源约束。其他的约束包括功能单元流水线化的程度，以及 ICU 和 EU 中其他资源的限制。例如，一个 Intel Pentium III 每个时钟周期只能解码三条指令。最后，分支预测逻辑的成功限制了处理器能够在指令流中超前工作以保持执行单元繁忙的程度。每次发生预测错误时，处理器从正确的位置重新开始都会引起很大的延迟。

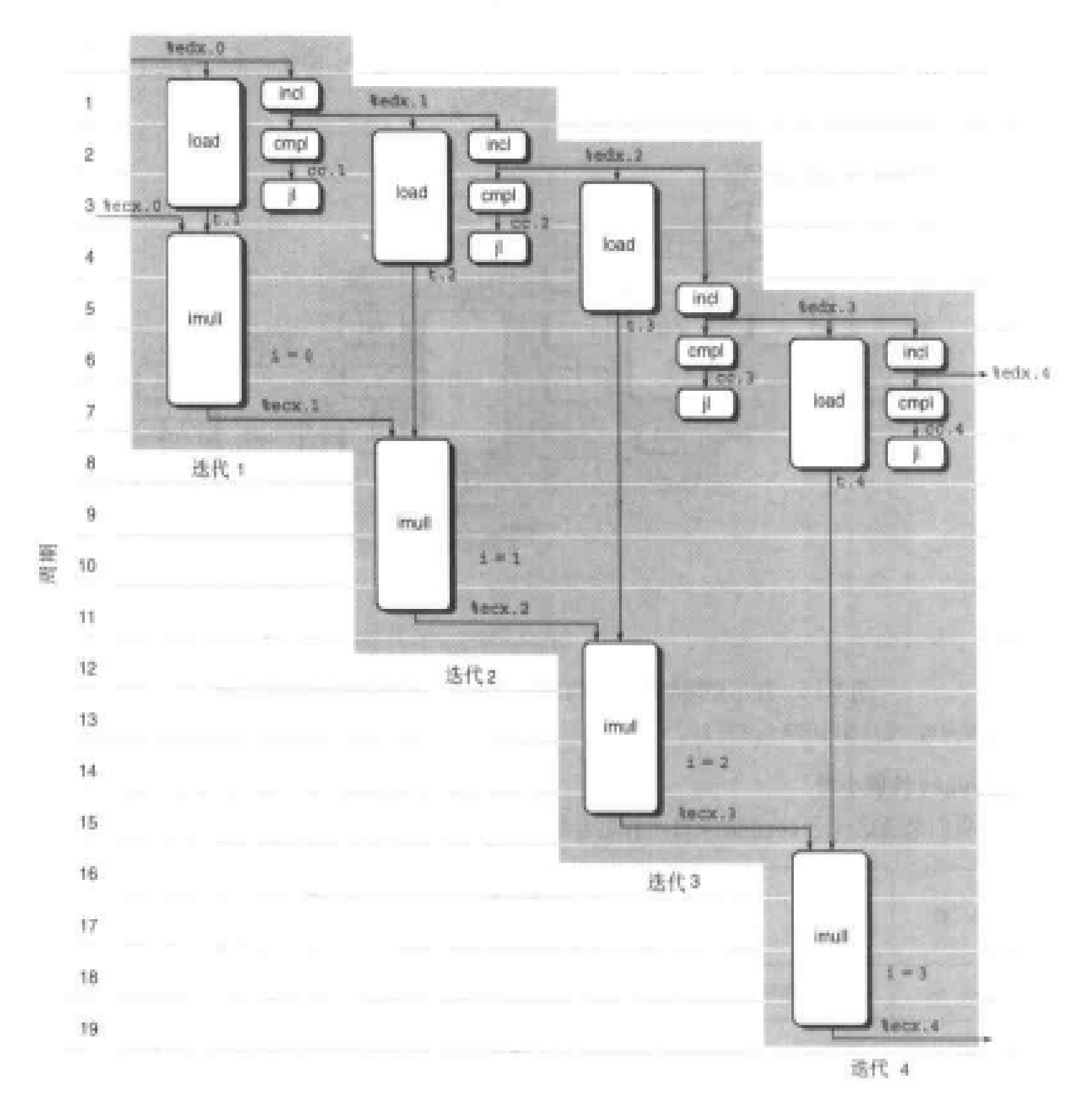


图 5.17 有实际资源约束的情况下，整数乘法的操作调度

乘法器的执行时间仍然是限制性能的因素。

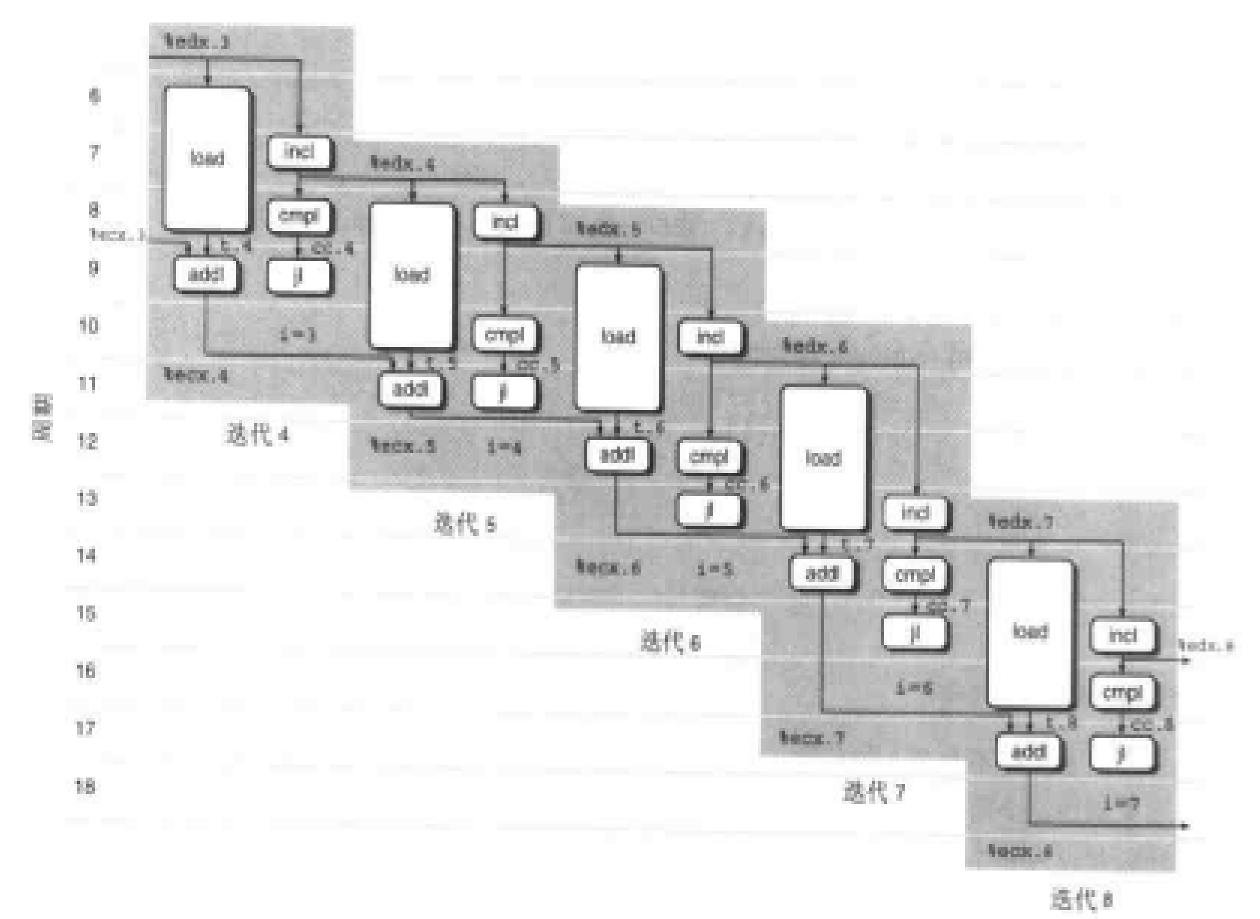


图 5.18 有实际资源约束的情况下，整数加法操作的调度

两个整数单元的限制将性能约束在 CPE 2.0。

5.8 降低循环开销

这样一个情况限制了整数加法的 combine4 的性能，那就是，每次迭代包括四条指令，但是只有两个功能单元能够执行这些指令。这四条指令中只有一条是对程序数据操作的。其他的都是计算循环的索引（loop index）和测试循环条件的循环开销的一部分。

我们可以通过在每次迭代中执行更多的数据操作来减小循环开销的影响，使用的是称为循环展开（loop unrolling）的技术。其思想是在一次迭代中访问数组元素并做乘法。这样得到的程序需要更少的迭代，从而降低了循环的开销。

图 5.19 给出了对我们的合并代码使用三次循环展开的版本。第一个循环一次处理数组的三个元素。也就是，循环索引 i 每次迭代会加 3，而一次迭代中会对数组元素 i、$i+1$ 和 $i+2$ 进行合并操作。

code/opt/combine.c

```
1   /* Unroll loop by 3 */
2   void combine5(vec_ptr v, data_t *dest)
3   {
4       int length = vec_length(v);
5       int limit = length-2;
6       data_t *data = get_vec_start(v);
```

```
7       data_t x = IDENT;
8       int i;
9
10      /* Combine 3 elements at a time */
11      for (i = 0; i < limit; i+=3) {
12          x = x OPER data[i] OPER data[i+1] OPER data[i+2];
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          x = x OPER data[i];
18      }
19      *dest = x;
20  }
```

code/opt/combine.c

图 5.19 展开循环三次

循环展开能减小循环开销的影响。

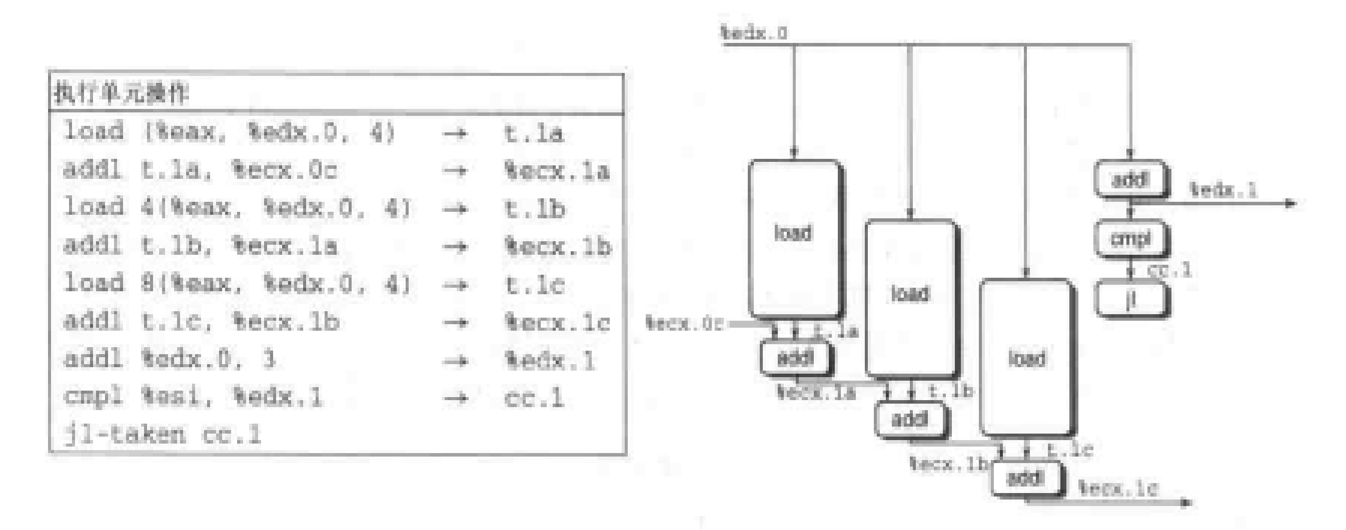
通常，向量长度不会是 3 的倍数。我们希望我们的代码对任意向量长度都能正常工作。我们从两个方面来解决这个需求。首先要确保第一次循环不会超出数组的界限。对于长度为 n 的向量，我们将循环限制设为 n-2。然后，我们会保证只有当循环索引 i 满足 $i < n-2$ 时才会执行这个循环，因此最大数组索引 $i+2$ 会满足 $i+2<(n-2)+2=n$。通常，如果循环展开 k 次，我们就把上限设为 $n-k+1$。那么最大循环索引 $i+k-1$ 会比 n 小。除此之外，我们加上第二个循环，以每次处理一个元素的方式处理向量的最后几个元素。这个循环体将会执行 0~2 次。

为了更好地理解带循环展开的代码的性能，让我们来看看内循环的汇编代码和它到操作的翻译：

汇编指令	执行单元操作		
`.L49:`			
`addl (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4)`	→	`t.1a`
	`addl t.1a, %ecx.0c`	→	`%ecx.1a`
`addl 4(%eax,%edx,4),%ecx`	`load 4(%eax, %edx.0, 4)`	→	`t.1b`
	`addl t.1b, %ecx.1a`	→	`%ecx.1b`
`addl 8(%eax,%edx,4),%ecx`	`load 8(%eax, %edx.0, 4)`	→	`t.1c`
	`addl t.1c, %ecx.1b`	→	`%ecx.1c`
`addl %edx,3`	`addl %edx.0, 3`	→	`%edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1`	→	`cc.1`
`jl .L49`	`jl-taken cc.1`		

正如前面提到的那样，循环展开本身只会帮助整数求和情况中代码的性能，因为我们的其他情况是被功能单元的执行时间限制的。对于整数求和，三次展开使得我们能够用六个整数/分支操作合并三个元素，如图 5.20 所示。用两个功能单元完成这些操作，我们潜在地能达到 CPE 1.0。图 5.21 表明，一旦我们到达迭代 3（i = 6），操作就会遵循一种规则的模式。迭代 4（i = 9）的操作有同样的时间安排，只不过移动了三个周期。这会真正得到 CPE 1.0。

我们对这个函数的测试表明 CPE 为 1.33，也就是说，每次迭代需要四个周期。很明显，我们在分析中未说明的某个资源约束延缓了计算，每次迭代要多一个周期。然而，比起未使用循环展开的代码，这个性能还是有改进的。



执行单元操作		
load (%eax, %edx.0, 4)	→	t.1a
addl t.1a, %ecx.0c	→	%ecx.1a
load 4(%eax, %edx.0, 4)	→	t.1b
addl t.1b, %ecx.1a	→	%ecx.1b
load 8(%eax, %edx.0, 4)	→	t.1c
addl t.1c, %ecx.1b	→	%ecx.1c
addl %edx.0, 3	→	%edx.1
cmpl %esi, %edx.1	→	cc.1
jl-taken cc.1		

图 5.20 三次展开整数加法的循环的第一次迭代的操作

使用这种程度的循环展开，我们可以用六个整数/分支操作合并三个数组元素。

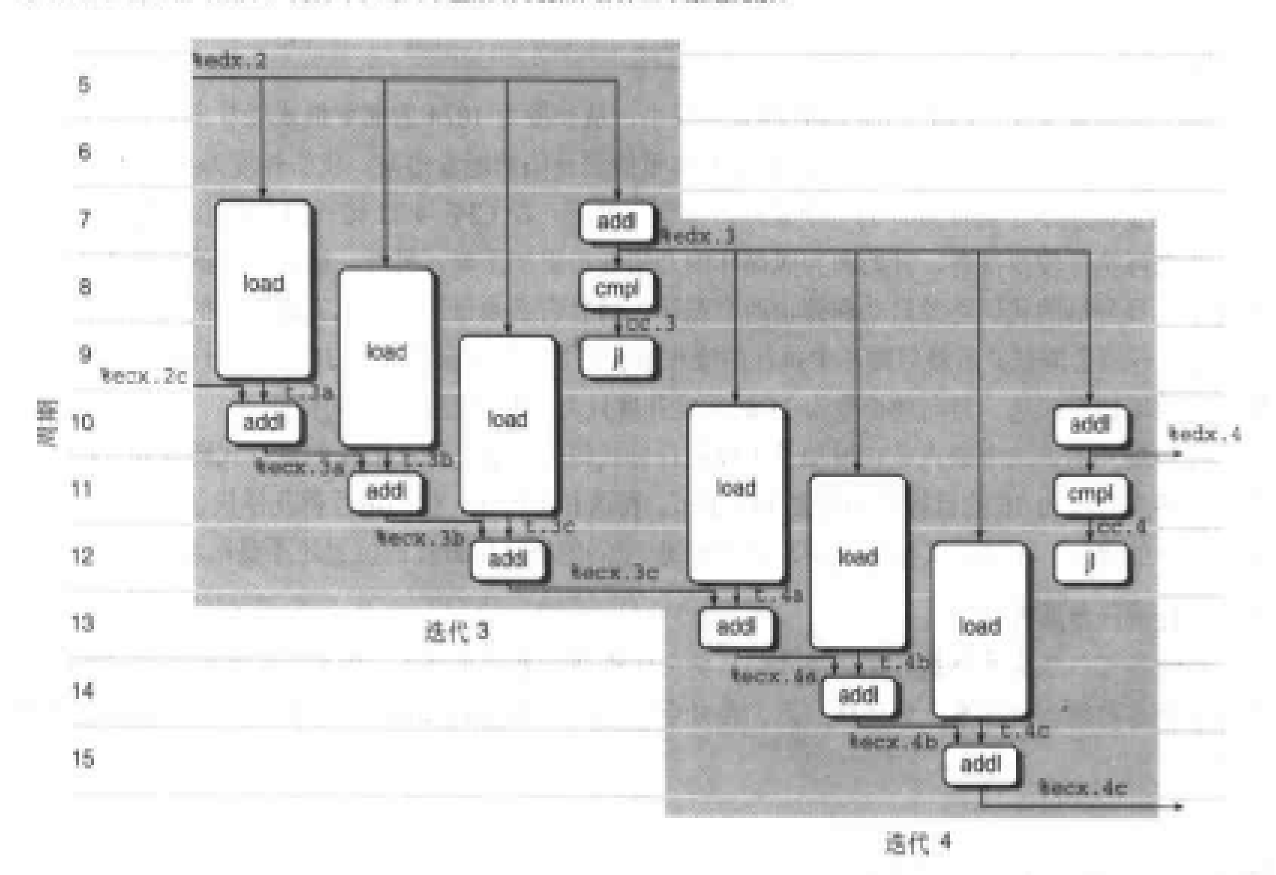


图 5.21 有限资源约束情况下，三次展开的整数求和操作的调度

原则上，这个过程可以达到 CPE 1.0，但是测量到的 CPE 为 1.33。

测量各种展开程度的性能，得到如下的 CPE 值：

向量长度	展开程度					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06

正如这些测量值表明的那样，循环展开能降低CPE。当循环展开度为2时，主循环的每次迭代需要三个时钟周期，得到CPE 3/2 = 1.5。当我们增加展开的度时，我们通常能获得更好的性能，接近CPE的理论极限1.0。注意到改进并非是单调的是很有意思的：展开度为3得到比展开度为4更好的性能。很明显，在后一种情况中，执行单元上操作的调度效率要低一些。

我们的CPE测量值不能解释开销因素，例如程序调用和准备循环的开销。使用循环展开，我们引入了一种新的开销——当向量长度不能被展开度整除时，需要完成所有剩下的元素。为了研究开销的影响，我们测量了各种向量长度的净CPE（net CPE）。净CPE是这样计算的，过程需要的总周期数除以元素的个数。对于不同展开度和两个不同的向量长度，我们获得下面的数据：

向量长度	展开程度					
	1	2	3	4	8	16
CPE	2.00	1.50	1.33	1.50	1.25	1.06
31 净CPE	4.02	3.57	3.39	3.84	3.91	3.66
1024 净CPE	2.06	1.56	1.40	1.56	1.31	1.12

对于长向量来说，CPE和净CPE的差别很小，从长度为1024的测量值就能看出来；但是对于短向量来说，影响就很明显，从长度为31的向量的测量值就能看出来。我们长度为31的向量的净CPE的测量值展示了循环展开的一个缺点。即使不展开，净CPE 4.02比长向量测出的2.06要高很多。当循环执行较少次时，开始和完成循环的开销变得更加重要。另外，循环展开的好处就不那么明显了。展开后的代码必须启动和停止两个循环，而且它必须每次一个地完成最后的元素。循环展开增加，开销会降低，而最后循环中执行的操作数会增加。当向量长度为1024时，性能通常会随着展开度的增加而改进。当向量长度为31时，展开度只为3时能得到最好的性能。

循环展开的第二个缺点是它增加了生成的目标代码的数量。combine4的目标代码需要63字节，但是循环展开度为16的目标代码需要142字节。在这种情况中，代码运行得几乎快了一倍，似乎要付出小小的代价。不过在其他情况中，这个时间-空间的折衷中最优的位置还不是很清楚。

旁注：让编译器展开循环

编译器可以很容易地执行循环展开。只要优化级别设置得足够高（例如，优化选项为“-O2”），许多编译器都能例行公事地做到这一点。在命令行上以“-funroll-loops”调用GCC，它会执行循环展开。

5.9 转换到指针代码

在进行下一步之前，我们应该再尝试一种有时能改进程序性能的转换，但这是以程序的可读性为代价的。C的一个独特的特性是能够对任意的程序对象创建和引用指针。实际上，指针运算与数组引用有很紧密的联系。表达式*(a+i)给出的指针运算和引用的组合正好等价于数组引用a[i]。有时，

我们能够通过使用指针而不是数组改进一个程序的性能。

图 5.22 给出了一个将过程 combine4 和 combine5 转换成指针代码的示例，分别得到过程 combine4p 和 combine5p。与保持指针 data 固定在向量的开始处相反，我们每次迭代时都移动它。然后，通过 data 的固定偏移量（0～2）来引用向量元素。最重要的是，我们能从过程中消除循环变量 i。为了确定循环该在什么时候中止，我们计算一个指针 dend 作为指针 data 的上界。比较这些过程和它们相应数组的过程的性能得到混合的结果：

函数	页数	方 法	整数		浮点数	
			+	*	+	*
combine4	335	累积在临时变量中	2.00	4.00	3.00	5.00
combine4p	351	指针版本	**3.00**	4.00	3.00	5.00
combine5	347	展开循环×3	1.33	4.00	3.00	5.00
combine5p	351	指针版本	1.33	4.00	3.00	5.00
combine5x4		展开循环×4	1.50	4.00	3.00	5.00
combine5px4		指针版本	**1.25**	4.00	3.00	5.00

对大多数情况来说，数组和指针版本的性能完全一样。不带展开的整数求和的指针版本的 CPE 实际上还变糟了一个周期。这个结果有点奇怪，因为指针和数组版本中的循环是非常类似的，如图 5.23 所示。很难想像为什么指针代码每次迭代需要多一个时钟周期。同样不可思议的是，过程四次循环展开的版本使用指针代码能产生每次迭代一个周期的性能提高，得到 CPE 1.25（每次迭代 5 个周期），而不是 1.5（每次迭代 6 个周期）。

code/opt/combine.c

```
1   /* Accumulate in local variable, pointer version */
2   void combine4p(vec_ptr v, data_t *dest)
3   {
4       int length = vec_length(v);
5       data_t *data = get_vec_start(v);
6       data_t *dend = data+length;
7       data_t x = IDENT;
8
9       for (; data < dend; data++)
10          x = x OPER *data;
11        *dest = x;
12   }
```

code/opt/combine.c

（a）combine4 的指针版本

code/opt/combine.c

```
1   /* Unroll loop by 3, pointer version */
2   void combine5p(vec_ptr v, data_t *dest)
3   {
4       data_t *data = get_vec_start(v);
```

```
5        data_t *dend = data+vec_length(v);
6        data_t *dlimit = dend-2;
7        data_t x = IDENT;
8
9        /* Combine 3 elements at a time */
10       for (; data < dlimit; data += 3) {
11           x = x OPER data[0] OPER data[1] OPER data[2];
12       }
13
14       /* Finish any remaining elements */
15       for (; data < dend; data++) {
16           x = x OPER data[0];
17       }
18       *dest = x;
19       }
```

code/opt/combine.c

(b) combine5 的指针版本

图 5.22 将数组代码转换成指针代码

在某些情况中，这能够导致性能的改进。

```
        combine4: type=INT, OPER = '+'
        data in %eax, x in %ecx, i in %edx, length in %esi
1    .L24:                              loop:
2      addl (%eax,%edx,4),%ecx          Add data[i] to x
3      incl %edx                        i++
4      cmpl %esi,%edx                   Compare i:length
5      jl .L24                          If <, goto loop
```

(a) Array code

```
        combine4p: type=INT, OPER = '+'
        data in %eax, x in %ecx, dend in %edx
1    .L30:                              loop:
2      addl (%eax),%ecx                 Add data[0] to x
3      addl $4,%eax                     data ++
4      cmpl %edx,%eax                   Compare data:dend
5      jb .L30                          If <, goto loop
```

(b) Pointer code

图 5.23 指针代码性能异常

虽然结构上两个程序非常相似，但是数组代码每次迭代需要 2 个周期，而指针代码需要 3 个。

根据我们的经验，指针和数组代码的相对性能依赖于机器、编译器，甚至于某个特殊的过程。我们已经看过编译器，它们对数组代码应用非常高级的优化，而对指针代码只应用最小限度的优化。为了可读性的缘故，通常数组代码更可取一些。

练习题 5.4

有时候，GCC 会自己将数组代码转换成指针代码。例如，使用整数数据和以加法作为合并操作时，GCC 为 combine5 的一个变种的内循环产生下列代码，使用的是八次循环展开：

```
1    .L6:
2       addl (%eax),%edx
3       addl 4(%eax),%edx
4       addl 8(%eax),%edx
5       addl 12(%eax),%edx
6       addl 16(%eax),%edx
7       addl 20(%eax),%edx
8       addl 24(%eax),%edx
9       addl 28(%eax),%edx
10      addl $32,%eax
11      addl $8,%ecx
12      cmpl %esi,%ecx
13      jl .L6
```

观察寄存器%eax 是如何每次迭代增加 32 的。

写出过程 combine5px8 的 C 代码，展示这段代码是如何计算指针、循环变量和中止条件的。按照图 5.19 的风格，给出使用任意数据和合并操作的通用格式。描述它与我们手写的指针代码（图 5.22）有何区别。

5.10 提高并行性

在此，我们的程序是受功能单元的执行时间限制的。不过，如图 5.12 中第三栏所示，处理器的几个功能单元是流水线化的，这意味着它们可以在前一个操作完成之前开始一个新的操作。我们的代码不能利用这种能力，即使是使用循环展开也不能，这是因为我们将累积值放在一个单独的变量 x 中。直到前面的计算完成之前，我们都不能计算 x 的新值。因此，处理器会暂停（stall），等待开始新的操作，直到当前操作完成。图 5.15 和图 5.17 中很清楚地展示了这个限制。即使有无限的处理器资源，乘法器也只能每四个时钟周期产生一个新的结果。对于浮点加法（三个周期）和乘法（五个周期）也会有类似的限制。

5.10.1 循环分割（loop splitting）

对于一个可结合和可交换的合并操作来说，比如说整数加法或乘法，我们可以通过将一组合并操作分割成两个或更多的部分，并在最后合并结果来提高性能。例如，P_n 表示元素 $a_0, a_1, \cdots, a_{n-1}$ 的乘积：

$$P_n = \prod_{i=0}^{n-1} a_i$$

假设 n 为偶数，我们还可以把它写成 $P_n = PE_n \times PO_n$，这里 PE_n 是索引值为偶数的元素的乘积，而 PO_n 是索引值为奇数的元素的乘积：

$$PE_n = \prod_{i=0}^{n/2-2} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-2} a_{2i+1}$$

图 5.24 展示的是使用这种方法的代码。它既使用了两次循环展开，以使每次迭代合并更多的元素，也使用了两路并行，将索引值为偶数的元素累积在变量 x0 中，而索引值为奇数的元素累积在变量 x1 中。同前面一样，我们还包括了第二个循环，对于向量长度不为 2 的倍数时，这个循环要累积所有剩下的数组元素。然后，我们对 x0 和 x1 应用合并操作，计算最终的结果。

code/opt/combine.c

```
1   /* Unroll loop by 2, 2-way parallelism */
2   void combine6(vec_ptr v, data_t *dest)
3   {
4       int length = vec_length(v);
5       int limit = length-1;
6       data_t *data = get_vec_start(v);
7       data_t x0 = IDENT;
8       data_t x1 = IDENT;
9       int i;
10
11      /* Combine 2 elements at a time */
12      for (i = 0; i < limit; i+=2) {
13          x0 = x0 OPER data[i];
14          x1 = x1 OPER data[i+1];
15      }
16
17      /* Finish any remaining elements */
18      for (; i < length; i++) {
19          x0 = x0 OPER data[i];
20      }
21      *dest = x0 OPER x1;
22  }
```

code/opt/combine.c

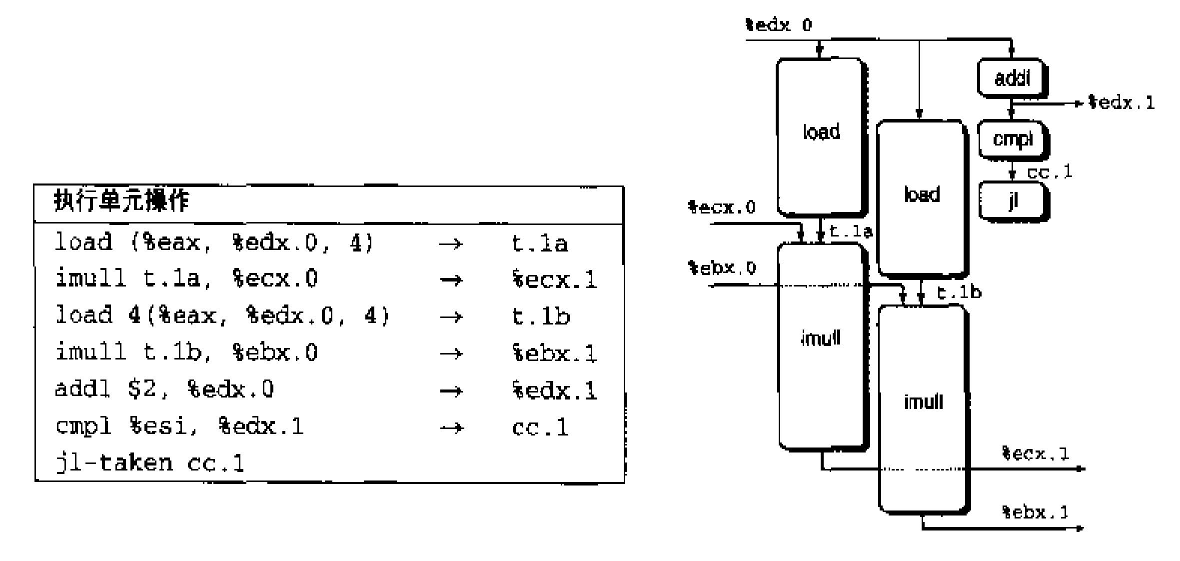
图 5.24 二次展开循环并使用二路并行

这种方法利用了功能单元的流水线能力。

为了了解这个代码是如何提高性能的，让我们来考虑对于整数乘法情况的循环到操作的翻译：

汇编指令	执行单元操作		
`.L151:`			
`imull (%eax,%edx,4),%ecx`	`load (%eax, %edx.0, 4)`	→	`t.1a`
	`imull t.1a, %ecx.0`	→	`%ecx.1`
`imull 4(%eax,%edx,4),%ebx`	`load 4(%eax, %edx.0, 4)`	→	`t.1b`
	`imull t.1b, %ebx.0`	→	`%ebx.1`
`addl $2,%edx`	`addl $2, %edx.0`	→	`%edx.1`
`cmpl %esi,%edx`	`cmpl %esi, %edx.1`	→	`cc.1`
`jl .L151`	`jl-taken cc.1`		

图 5.25 给出了第一次迭代（i=0）的操作的图形化表示。正如这张表说明的那样，循环中的两个乘法是相互独立的。一个以寄存器%ecx 作为它的源和目的（对应于程序变量 x0），而另一个以寄存器%ebx 作为它的源和目的（对应于程序变量 x1）。第二个乘法在第一个的后一个的后一个周期就可以开始了。这利用了加载单元和整数乘法器的流水线化的能力。



执行单元操作		
`load (%eax, %edx.0, 4)`	→	`t.1a`
`imull t.1a, %ecx.0`	→	`%ecx.1`
`load 4(%eax, %edx.0, 4)`	→	`t.1b`
`imull t.1b, %ebx.0`	→	`%ebx.1`
`addl $2, %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

图 5.25 二次展开、二路并行的整数乘法内循环的第一次迭代操作

两个乘法操作是逻辑上独立的。

图 5.26 给出的是整数乘法的头三次迭代（i=0，2 和 4）的图形化表示。对于每次迭代，两个乘法都必须等待，直到前一次迭代的结果计算出来。这个机器还是能每四个时钟周期产生两个结果，得到了理论上的 CPE 2.0。在这幅图中，我们不考虑整数功能单元的有限集合，但是也没有证明它是这个特殊过程的限制。

比较只进行循环展开和使用循环展开以及两路并行，我们得到以下的性能：

函 数	页数	方 法	整 数		浮点数	
			+	*	+	*
		展开 ×2	1.50	4.00	3.00	5.00
combine6	354	展开 ×2，并行 ×2	1.50	2.00	2.00	2.50

对于整数求和，并行化并没有帮助，因为整数加法的执行时间只是一个时钟周期。不过对于整数和

浮点乘法，我们将 CPE 减小了 1 倍。从本质上看，我们加倍地使用了功能单元。对于浮点加法，某些其他的资源约束将我们的 CPE 限制在了 2.0，而不是理论值 1.5。

我们早就知道，二进制补码运算是可交换和可结合的，甚至于溢出时也是如此。因此，对于整数数据类型，在所有可能的情况下，combine6 计算出的结果都和 combine5 计算出的相同。因此，优化编译器潜在地能够将 combine4 中所示的代码首先转换成 combine5 的一个二路循环展开变种，然后再通过引入并行性，将之转换成 combine6 的一个变种，在优化编译器的语言中，这称为迭代分割（iteration splitting）。许多编译器自动进行循环展开，但是进行迭代分割的编译器相对比较少了。



图 5.26 在有限资源情况下，二次展开、二路并行的整数乘法操作的调度

乘法器现在可以每 4 个周期产生两个值了。

另一方面，我们知道浮点乘法和加法不是可结合的。因此，由于四舍五入或溢出，combine5 和 combine6 可能产生不同的结果。例如，假想这样一种情况，所有索引值为偶数的元素都是绝对值非常大的数，而索引值为奇数的元素都非常接近于 0.0。那么，即使最终的乘积 P_n 不会溢出，乘积 PE_n 也可能溢出，或者 PO_n 也可能下溢。不过在大多数现实的程序中，不太可能出现这样的情况。因为大多数物理现象是连续的，所以数字数据也趋向于相当的平滑，不会出什么问题。即使是有不连续

的时候，它们通常也不会导致前面描述的条件那样的周期性模式。按照严格顺序对元素求和不太可能会有比“分成两组独立求和，然后再将这两个和相加”根本上更好的准确性。对大多数应用程序来说，使性能翻倍要比对奇怪的数据模式产生不同的结果的风险更重要。但是，程序开发人员应该与潜在的用户协商，看看是否有特殊的条件，可能会导致修改后的算法不能接受。

就像我们能以任意次数 k 展开循环，我们也可以增加并行度为任意因子 p，使得 k 可以被 p 整除。下面是对于不同展开次数和并行度的一些结果：

方　法	整　数		浮点数	
	+	*	+	*
展开度×2	1.50	4.00	3.00	5.00
展开度×2，并行度×2	1.50	2.00	2.00	2.50
展开度×4	1.50	4.00	3.00	5.00
展开度×4，并行度×2	1.50	2.00	1.50	2.50
展开度×8	1.25	4.00	3.00	5.00
展开度×8，并行度×2	1.25	2.00	1.50	2.50
展开度×8，并行度×4	1.25	1.25	1.61	2.00
展开度×8，并行度×8	1.75	1.87	1.87	2.07
展开度×9，并行度×3	1.22	1.33	1.66	2.00

正如这张表说明的那样，增加循环的展开度和并行度能帮助程序性能达到某个点，但是当取到极限的时候，性能的增长就减缓了。在下一节中，我们将会描述出现这种现象的两个原因。

5.10.2 寄存器溢出（register spilling）

循环并行性的好处是受描述计算的汇编代码的能力限制的。特别地，IA32 指令集只有很少量的寄存器来存放累积的值。如果我们有并行度 p 超过了可用的寄存器数量，编译器会诉诸于溢出（spilling），将某些临时值存放到栈中。一旦出现这种情况，性能会急剧下降。当我们试图使 p=8 时，对我们的基准程序就发生了这种情况。我们的测量值显示此种情况下的性能比 p=4 时的性能更差。

对于整数数据类型的情况，总共只有八个整数寄存器可用。其中有两个（%ebp 和%esp）指向栈中的区域。在这段代码的指针版本中，剩下的六个寄存器中有一个要存放指针 data，还有一个要存放停止位置 dend。这就只剩下四个整数寄存器可以用来存放累积的值了。在这段代码的数组版本中，我们需要三个寄存器来保存循环索引值 i、停止索引值 limit，以及数组地址 data。这就只剩下三个整数寄存器可以用来存放累积的值。对于浮点数据类型，我们需要八个寄存器中的两个来保存中间值，剩下六个用于累积值。因此，我们能得到在发生寄存器溢出之前，最大并行度为 6。

八个整数和八个浮点寄存器的限制是 IA32 指令集的不幸产物。前面讲到过的重命名机制消除了寄存器名字和寄存器数据实际位置之间的联系。在现代处理器中，寄存器名字只简单地用来标识在功能单元之间传递的程序值。IA32 只提供了很少量的这样的标识符，限制了在程序中能表达的并行性的数量。

通过检查汇编代码就能发现溢出的发生。例如，在八路并行的代码的第一个循环中，我们看到下面的指令序列：

```
type=INT, OPER = '*'
x6 in -12(%ebp), data+i in %eax
```

```
1    movl -12(%ebp),%edi          Get x6 from stack
2    imull 24(%eax),%edi          Multiply by data[i+6]
3    movl %edi,-12(%ebp)          Put x6 back
```

在这段代码中，一个栈的位置被用来存放x6，八个局部变量中的一个被用来累积和。代码将这个值加载到一个寄存器中，将它乘以一个数据元素，然后再存回到同一个栈的位置中。作为一条通用原则，无论何时当一个编译了的程序显示出在某个频繁使用的内循环中有寄存器溢出的迹象时，它都会倾向于重写代码，使之需要较少的临时值。通过减少局部变量的数量能够做到这一点。

练习题 5.5

下面给出的是根据combine6的一个变种产生的代码，它使用了八次循环展开和四路并行。

```
1   .L152:
2     addl (%eax),%ecx
3     addl 4(%eax),%esi
4     addl 8(%eax),%edi
5     addl 12(%eax),%ebx
6     addl 16(%eax),%ecx
7     addl 20(%eax),%esi
8     addl 24(%eax),%edi
9     addl 28(%eax),%ebx
10    addl $32,%eax
11    addl $8,%edx
12    cmpl -8(%ebp),%edx
13    jl .L152
```

A. 什么程序变量被溢出放到了栈中？

B. 放到了栈中的什么位置？

C. 为什么将那个溢出值放到栈中是好的选择呢？

使用浮点数据时，我们希望将所有的局部变量都放在浮点寄存器栈中。我们还需要保持栈顶可用于从存储器加载数据。这限制了并行度小于或等于7。

5.10.3 对并行的限制

对于我们的基准程序，主要的性能限制是由于功能单元的能力。如图5.12所示，整数乘法器和浮点加法器只能每个时钟周期发起一条新操作。这，加上对加载单元的类似限制，将这些情况的CPE限制在了1.0。浮点乘法器只能每两个时钟周期发起一条新操作。这就将这种情况的CPE限制在了2.0。由于加载单元的限制，整数求和被限制在了CPE 1.0。这就导致了下面对达到的性能与理论极限之间的比较：

方法	整数		浮点数	
	+	*	+	*
实际	1.06	1.25	1.50	2.00
理论限制	1.00	1.00	1.00	2.00

在这张表中，为每种情况，我们都选择能达到最佳性能的展开和并行的组合。对于整数求和和乘积以及浮点乘积，我们能够接近理论极限值。某个（或某些）与机器相关的因素将浮点乘法能达到的 CPE 限制在了 1.50，而不是理论极限值 1.0。

练习题 5.6

考虑下面的计算 n 个整数的数组乘积的函数。我们三次展开这个循环。

```
int aprod(int a[], int n)
{
    int i, x, y, z;
    int r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Product computation */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

对于标号为 Product computation 的行，我们可以用括号创建出计算的五个不同的结合，如下所示：

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

我们在 Pentium III 上 测试了函数的这五个版本。回想一下图 5.12，在这种机器上整数乘法操作的执行时间为 4 个周期，发射时间为 1 个周期。

下面的表给出了一些 CPE 的值，也漏掉了一些。测量出的 CPE 值是实际观测到的。“理论 CPE”的意思是当限制因素只为执行时间和整数乘法器时能够达到的性能。

版本	测量出的 CPE	理论 CPE
A1	4.00	
A2	2.67	
A3		
A4	1.67	
A5		

填写出漏掉的条目。对于漏掉的 CPE 的度量值，你可以使用来自其他有相同计算行为的版本的值。对于 CPE 的理论值，你可以只考虑乘法器的执行时间和发射时间，确定一次迭代所需的周期数，然后再除以 3。

5.11 综合：优化合并（Combing）代码的效果小结

现在，我们已经考虑了合并（Combing）代码的六个版本，其中有的还有多个变种。让我们暂停一下，来看看这种努力的整体效果，以及我们的代码是如何在一台不同的机器上执行的。图 5.27 给出的是对我们所有函数以及几个其他变种的度量性能。正如看到的那样，我们只要简单地展开循环多次，就能达到整数求和的最大性能，但是对于其他操作，我们引入一些（但不是很多）并行性。整体性能达到了 27.6 倍，比我们原始的代码好了很多。

函数	页数	方法	整数 +	整数 *	浮点数 +	浮点数 *
combine1	329	未优化的抽象的	42.06	41.86	41.44	160.00
combine1	329	抽象的 -O2	31.25	33.25	31.25	143.00
combine2	330	移动 vec_length	20.66	21.25	21.15	135.00
combine3	334	直接数据访问	6.00	9.00	8.00	117.00
combine4	335	累积在临时变量中	2.00	4.00	3.00	5.00
combine5	347	展开×4	1.50	4.00	3.00	5.00
		展开×16	**1.06**	4.00	3.00	5.00
combine6	354	展开×2，并行×2	1.50	2.00	2.00	2.50
		展开×4，并行×2	1.50	2.00	1.50	2.50
		展开×8，并行×4	1.25	**1.25**	**1.50**	**2.00**
最坏：最好			39.7	33.5	27.6	80.0

图 5.27 所有合并函数的结果比较

性能最好的版本用粗体表示。

5.11.1 浮点性能异常

图 5.27 最引人注目的特性之一是，当我们从将 combine3 的乘积累积在存储器中，到将 combine4 的乘积累积在一个浮点寄存器中，浮点乘法的周期数急剧下降。通过这么一点小小的改动，代码运行就快了 23.4 倍。当出现这样一种出乎意料的结果时，猜测是什么可能引起这样的行为，然后设计一系列试验来评估这个假设，这是很重要的。

当我们查看这张表时，对浮点乘法的情况来说，如果将结果累积在存储器中，好像有点奇怪的事情会发生。即使功能单元的周期数是相当的，它的性能比浮点加法或整数乘法的性能都要差很多。在一个 IA32 处理器上，所有的浮点操作都是以扩展的 80 位精度执行的，而浮点寄存器也是按照这个格式存储值的。只有当寄存器中的值写入存储器中时，才把它转换成 32 位（浮点数）或 64 位（双精度）格式。

检查我们测试所用的数据，问题的根源就很清楚了。测试是在一个长度为 1024 的向量上执行的，这个向量的每个元素 i 的值等于 $i+1$。因此，我们是在试图计算 1024!，它大约是 5.4×10^{2639}。这样大的一个数可以用扩展精度的浮点格式（它可以表示到大约 10^{4932} 的数）表示，但是它大大超出了单精度（大约 10^{38}）或双精度（大约 10^{308}）能表示的范围。当我们到达 i=34 时，单精度的情况就会溢出了，而当我们到达 i=171 时，双精度的情况就会溢出了。一旦我们达到这一点，每次执行 combine3 的内部循环中的语句

```
*dest = *dest OPER val;
```

都要从 dest 中读值+∞，把它乘以 val，得到+∞，然后将它存回到 dest。很明显，这个计算的某个部分需要比浮点乘法所要的正常的五个时钟周期长得多的时间。实际上，对这个操作进行测试，我们发现用一个数乘以无穷大会花费 110～120 个周期。很可能，硬件察觉了这个特殊情况，发出一个陷阱（trap），它会使一个软件函数执行实际的计算。CPU 设计者觉得这样的情况会非常罕见，以至于他们不需要用硬件设计的一部分来处理它。对于下溢也会发生类似的行为。

当我们在每个向量元素等于 1.0 的数据上运行基准程序时，对双精度和单精度，combine3 的 CPE 都达到了 10.00 周期。这与对其他数据类型和操作进行度量到的时间更加一致，而且与 combine4 的时间也是相当的。

这个示例说明了评估程序性能的一个挑战，原本看上去无足轻重的数据和操作条件能严重地影响测量结果。

5.11.2 变换平台

虽然我们是在一个特殊的机器和编译器环境中讲述我们的优化策略的，但是通用的原则也适用于其他机器和编译器。当然，最优的策略可能是与机器相关的。作为一个示例，图 5.28 给出的是 Compaq Alpha 21164 处理器在与图 5.27 中所示的 Pentium III 相当的条件下的性能结果。这些测试采用的是 Compaq C 编译器生成的代码，它使用了比 GCC 更多的高级优化。我们观察到，随着我们沿着表往下走，周期时间通常会降低，就像对其他机器一样。我们看到，我们能有效地运用更高程度（八路）的并行，这是因为 Alpha 有 32 个整数和 32 个浮点寄存器。正如这个例子说明的那样，程序优化的通用原则对各种不同的机器都适用，即使某种特殊的特性组合会导致最优性能依赖于特殊的机器。

函数	页数	方法	整数		浮点数	
			+	*	+	*
combine1	329	未优化的抽象的	40.14	47.14	52.07	53.71
combine1	329	抽象的 -O2	25.08	36.05	37.37	32.02
combine2	330	移动 vec_length	19.19	32.18	28.73	32.73
combine3	334	直接数据访问	6.26	12.52	13.26	13.01
combine4	335	累积在临时变量中	1.76	9.01	8.01	8.01
combine5	347	展开×4	1.51	9.01	6.32	6.32
		展开×16	1.25	9.01	6.33	6.22
combine6	354	展开×4，并行×2	1.19	4.69	4.44	4.45
		展开×8，并行×4	1.15	**4.12**	2.34	**2.01**
		展开×8，并行×8	**1.11**	4.24	**2.36**	2.08
最坏：最好			36.2	11.4	22.3	26.7

图 5.28 所有合并函数运行在 Compaq Alpha 21164 处理器上的结果比较

同样的通用优化技术在这种机器上也有用。

5.12 分支预测和预测错误处罚

正如我们前面提到过的，现代处理器在执行当前指令之前能工作得很好，从存储器读新指令，并解码指令，以确定在什么操作数上执行什么操作。只要指令遵循的是一种简单的顺序，那么这种指令流水线化（instruction pipeline）就能很好地工作。不过，当遇到分支的时候，处理器必须猜测分支该往哪个方向走。对于条件转移的情况，这意味着要预测是否会选择分支。对于像间接跳转（就像我们在代码中看到的，跳转到由一个跳转表条目指定的地址）或过程返回这样的指令，这意味着预测目标地址。在这里，我们集中讨论条件分支。

在一个使用投机执行（speculative execution）的处理器中，处理器会开始执行预测的分支目标处的指令。它这样做的方式是，避免修改任何实际的寄存器或存储器位置，直到确定了实际的结果。如果预测是正确的，处理器就简单地“提交”投机执行的指令的结果，把它们存储到寄存器或存储器中。如果预测是错误的，处理器必须丢弃掉所有投机执行的结果，在正确的位置，重新开始取指令的过程。这样做会引起很大的分支处罚（branch penalty），因为在产生有用的结果之前，必须重新填充指令流水线。

直到最近，支持投机执行所需的技术都被认为是开销太大，对除了最高级的超级计算机以外的所有机器来说都是异乎寻常的。不过大约从 1998 年开始，集成电路技术使得可以在一块芯片上放置如此之多的电路，以至于有些电路可以专门用来支持分支预测和投机执行。到目前，台式机或服务器中几乎每个处理器都支持投机执行。

在优化我们的合并过程中，我们没有看到循环结构对性能的任何限制。也就是，看上去对性能惟一的限制因素是由于功能单元。对于这个过程处理，处理器通常能够预测循环结尾处的分支的方向。实际上，如果处理器总是预测会选择分支，那么除了对最后一次迭代以外，它都是对的。

人们已经提出了许多方法来预测分支，而且对这些方法的性能也进行了很多研究。一种常见的启发式方法是预测任意到较低地址的分支都会被选择，而任何到较高地址的分支则不会。到较低地址的分支是用来关闭循环的，因为循环通常会执行多次，预测这些分支会被选择是个好主意。另一方面，前向分支是用于条件计算的。实验表明后向选择、前向不选择的启发式方法在大约 65%的时间里是正确的。而预测所有的分支都会被选择的成功率只为大约 60%。也有更加复杂的策略，需要更多的硬件。例如，Intel Pentium II 和 III 处理器使用的分支预测策略声称在 90%～95%的时间里都是正确的。

我们可以进行实验来测试处理器的分支预测的能力，以及预测错误的代价。我们用图 5.29 中所示的绝对值函数作为我们的测试示例。这幅图还给出了编译后的形式。对于非负的参数，分支会被选择，以略过为负时的指令。我们对这个计算一个数组中每个元素绝对值的函数计时，这个数组是由+1 和-1 的各种模式组成的。对于规则的模式（例如，全+1、全-1 或交替的+1 和-1），我们发现函数需要 13.01～13.41 个周期。我们以此作为我们完美分支条件下性能的估计值。对于一个设置为+1 和-1 的随机模式的数组，我们发现函数需要 20.32 个周期。随机处理的一个原则是无论用什么策略来猜测值的序列，如果底层的处理是真正随机的，那么我们只可能有 50%的时间是正确的。例如，无论一个人用什么策略来猜扔硬币的结果，只要扔硬币是公平的，成功的概率就只能是 0.5。因而，我们可以看到，这个处理器预测错误的分支会引起大约 14 个时钟周期的处罚，因为 50%的预测错误率会导致函数运行平均慢 7 个周期。意思就是说，对 absval 的调用依据分支预测的成功率，需要

13～27 个周期。

code/opt/absval.c

```
1   int absval(int val)
2   {
3       return (val<0) ? -val : val;
4   }
```

code/opt/absval.c

(a) C 代码

```
1    absval:
2      pushl %ebp
3      movl %esp,%ebp
4      movl 8(%ebp),%eax     Get val
5      testl %eax,%eax       Test it
6      jge .L3               If >0, goto end
7      negl %eax             Else, negate it
8    .L3:               end:
9      movl %ebp,%esp
10     popl %ebp
11     ret
```

(b) 汇编代码

图 5.29 绝对值代码

我们用这段代码来测量分支预测错误的代价。

14 个周期的处罚是相当大的。例如，如果我们的预测准确率只有 65%，那么对每个分支指令，处理器平均会浪费 14×0.35=4.9 周期。即使是 Pentium II 和 III 声称的预测准确率是 90%～95%，由于预测错误，每个分支都会浪费大约 1 个周期。对实际程序的研究表明，在典型的“整数”程序（也就是，那些不处理小数数据的程序）中，分支大约占到了所有执行指令的 15%，而在典型的小数程序中，分支大约占 3%～12% [33]。因此，由于低效率的分支处理造成的任何时间浪费都能对处理器性能产生很大的影响。

许多与数据相关的分支是根本不能预测的。例如，没有任何依据猜测我们绝对值函数的一个参数是正数还是负数。为了提高包括条件求值代码的性能，许多处理器设计被扩展来包括条件传送（conditional move）指令。这些指令允许某些形式的条件句不需要任何分支语句就能实现。

在 IA32 指令集中，从 PentiumPro 开始增加了许多不同的 cmov 指令。最近所有的 Intel 和与 Intel 兼容的处理器都支持这些指令，它们执行的操作类似于 C 代码：

```
if (COND)
    x = y;
```

这里 y 是源操作数，而 x 是目的操作数。条件 COND 确定是否要执行拷贝操作，它是基于条件码值的某种组合的，类似于测试和条件转移指令。作为一个示例，当条件码表明一个值小于 0 时，cmovll 指令执行一个拷贝。注意，这条指令的第一个“l”表示“less（小于）”，而第二个“l”是 GAS 表示长字的后缀。

下面的汇编代码展示了如何用条件传送来实现绝对值：

```
1    movl 8(%ebp),%eax          Get val as result
2    movl %eax,%edx             Copy to %edx
3    negl %edx                  Negate %edx
4    testl %eax,%eax            Test val
5    Conditionally move %edx to %eax
6    cmovll %edx,%eax           If < 0, copy %edx to result
```

正如这段代码表明的那样，策略是将 val 设置为返回值，计算 -val，并当 val 为负时，有条件地将它传送到寄存器%eax 以改变返回值。我们对这段代码的测试表明无论数据模式怎样，它都运行 13.7 个周期。该整体性能明显地好于需要 13～27 个周期的过程。

练习题 5.7

你的一个朋友写了一个利用条件传送指令的优化编译器。你试着编译下面的 C 代码：

```
1    /* Dereference pointer or return 0 if null */
2    int deref(int *xp)
3    {
4        return xp ? *xp : 0;
5    }
```

编译器为过程体产生下面的代码。

```
1    movl 8(%ebp),%edx     Get xp
2    movl (%edx),%eax      Get *xp as result
3    testl %edx,%edx       Test xp
4    cmovll %edx,%eax      If 0, copy 0 to result
```

解释一下为什么这段代码提供的不是 deref 的合法实现。

GCC 目前的版本不用条件传送来产生任何代码。由于期望与以前的 486 和 Pentium 处理器保持兼容，编译器不利用这些新特性。在我们的试验中，我们使用的是上面所示的手写的汇编代码。由于代码生成的质量更糟糕，一个使用 GCC 工具在 C 程序中嵌入汇编代码的版本需要 17.1 个周期。

不幸的是，C 程序员对改进一个程序的分支性能是无能为力的，除了意识到数据相关的分支会引起性能上很高的花费。除此之外，程序员对编译器产生的详细的分支结构几乎没有什么控制，很难使分支更容易预测一些。最终，我们必须依靠两种因素的结合：一是编译器生成好的代码，尽量减少条件分支的使用；另一个是处理器有效地分支预测，降低分支预测错误的数量。

5.13 理解存储器性能

到目前为止我们写的所有代码，以及我们运行的所有测试，对存储器的需求都相对较少。例如，我们都是在长度为 1024 的向量上测试那些合并函数，数据量不会超过 8096 字节。所有的现代处理器都包含一个或多个高速缓存（cache）存储器，以提供对这样少量的存储器的快速访问。图 5.12

中所有的计时都假设被读或写的数据是在高速缓存中的。在第 6 章中，我们会更详细地探究高速缓存是如何工作的，以及如何编写充分利用高速缓存的代码。

在本节中，我们会进一步研究加载和存储操作的性能，我们仍然假设被读或写的数据是在高速缓存中的。如图 5.12 所示，这两个单元的执行时间都为 3，而发射时间为 1。迄今为止我们的所有程序都只用了加载操作，都有这样一个属性，一条加载操作的地址依赖于对某个寄存器执行增加操作，而不是依赖于另一条加载操作的结果。因此，如图 5.15～图 5.18、图 5.21 和图 5.26 所示，加载操作能利用流水线化，每个时钟周期开始新的加载操作。加载操作相对较长的执行时间对程序性能没有任何负面影响。

5.13.1 加载的执行时间

作为一个性能受加载操作执行时间限制的代码示例，考虑函数 list_len，如图 5.30 所示。这个函数计算的是一个链表的长度。在该函数的循环中，变量 ls 的每个连续的值都依赖于指针引用 ls->next 读出的值。我们的测试表明函数 list_len 的 CPE 为 3.0，我们认为这是加载操作执行时间的直接反映。为了说明这一点，来考虑这个循环的汇编代码，以及它的第一次迭代到操作的翻译：

汇编指令	执行单元操作		
L27:			
incl %eax	incl %eax.0	→	%eax.1
movl (%edx),%edx	load (%edx.0)	→	%edx.1
testl %edx,%edx	testl %edx.1,%edx.1	→	cc.1
jne .L27	jne-taken cc.1		

code/opt/list.c

```
1    typedef struct ELE {
2        struct ELE *next;
3        int data;
4    } list_ele, *list_ptr;
5
6    int list_len(list_ptr ls)
7    {
8        int len = 0;
9
10       for (; ls; ls = ls->next)
11           len++;
12       return len;
13   }
```

code/opt/list.c

图 5.30 链表函数

这举例说明了加载操作的执行时间。

寄存器%edx 的每个连续的值都依赖于一个以%edx 作为操作数的加载操作的结果。图 5.31 给出的是这个函数头三次迭代的操作的调度。正如看到的那样，加载操作的执行时间将 CPE 限制在了 3.0。

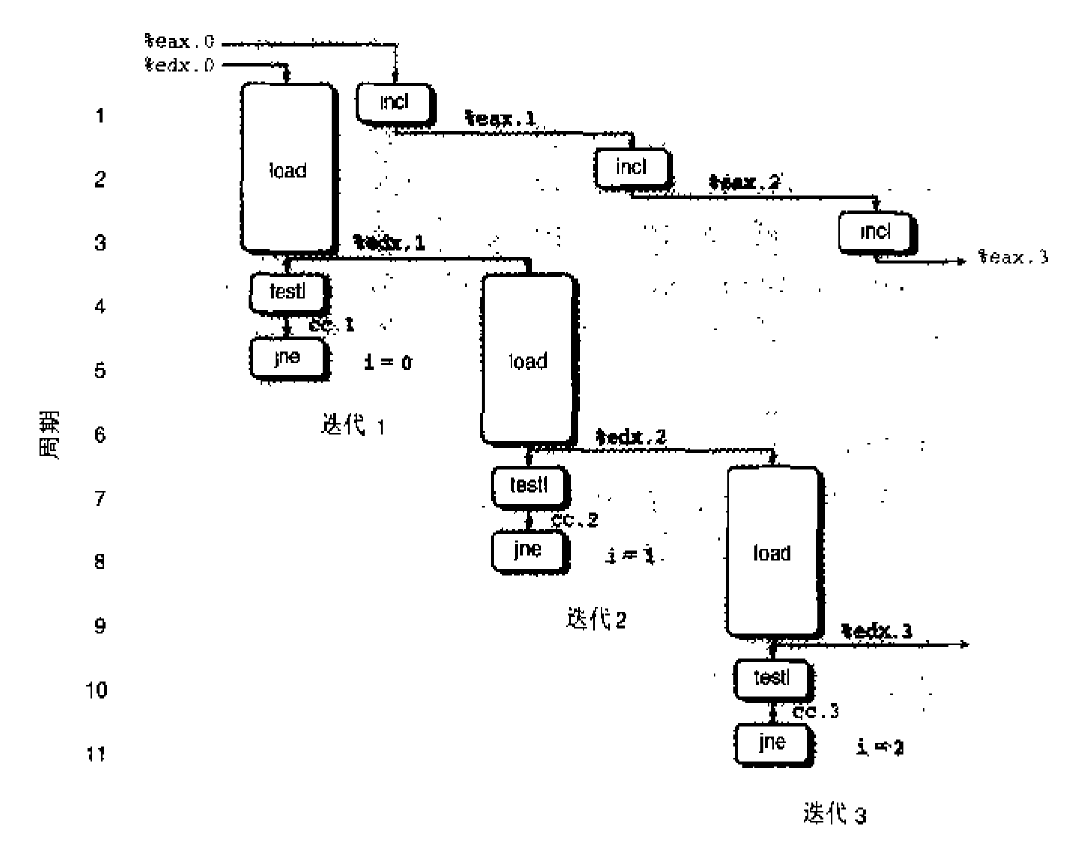


图 5.31 链表长度函数的操作的调度

加载操作的执行时间将 CPE 的最小值限制在了 3.0。

5.13.2 存储的执行时间

在迄今为止我们所有的示例中，我们只通过使用加载操作从一个存储器位置读数据到一个寄存器中来与存储器交互。与之对应的，存储（store）操作将一个寄存器值写到存储器。正如图 5.12 表明的那样，这个操作名义上的执行时间也是三个周期，发射时间为一个周期。不过，它的行为以及它与加载操作的交互有几个微妙的问题。

code/opt/copy.c

```
1   /* Set element of array to 0 */
2   void array_clear(int *src, int *dest, int n)
3   {
4       int i;
5
6       for (i = 0; i < n; i++)
7           dest[i] = 0;
8   }
9
10  /* Set elements of array to 0, unrolling by 8 */
11  void array_clear_8(int *src, int *dest, int n)
12  {
13      int i;
14      int len = n - 7;
15
```

```
16      for (i = 0; i < len; i+=8) {
17          dest[i] = 0;
18          dest[i+1] = 0;
19          dest[i+2] = 0;
20          dest[i+3] = 0;
21          dest[i+4] = 0;
22          dest[i+5] = 0;
23          dest[i+6] = 0;
24          dest[i+7] = 0;
25      }
26      for (; i < n; i++)
27          dest[i] = 0;
28  }
```
code/opt/copy.c

图 5.32 清空数组的函数

这说明了存储操作的流水线化。

与加载操作一样，在大多数情况中，存储操作能够在完全流水线化的模式中工作，每个周期开始一条新的存储。例如，考虑图 5.32 中所示的函数，它们将一个长度为 n 的数组 dest 的元素设置为 0。我们对这第一个版本的测试表明 CPE 为 2.00。因为每次迭代都需要一个存储操作，所以很明显处理器至少每 2 个周期能够开始一条新的存储操作。为了进一步探究，我们试着展开这个循环八次，如 array_clear_8 的代码所示。对于这个函数，我们测量得到 CPE 1.25。也就是，每次迭代需要大约 10 个周期，并发射 8 个存储操作。因此，我们几乎已经达到每个周期一条新存储操作的最优极限了。

同到目前为止我们已经考虑过的其他操作不同，存储操作并不影响任何寄存器值。因此，就其本性来说，一系列存储操作一定是相互独立的。实际上，只有一条加载操作是受一条存储操作结果影响的，因为只有一条加载操作能从由存储操作写的那个存储器位置读回值。图 5.33 所示的函数 write_read 说明了加载和存储操作之间可能的相互影响。这幅图也展示了该函数的两个示例执行，是对两元素数组 a 调用的，该数组的初始内容为-10 和 17，参数 cnt 等于 3。这些执行说明了加载和存储操作的一些微妙之处。

在图 5.33 的示例 A 中，参数 src 是一个指向数组元素 a[0]的指针，而 dest 是一个指向数组元素 a[1]的指针。在此种情况中，指针引用*src 的每次加载都会得到值-10。因此，在两次迭代之后，数组元素就会分别保持固定为-10 和-9。从 src 读出的结果不受对 dest 的写的影响。在较大次数的迭代上测试这个示例得到 CPE 2.00。

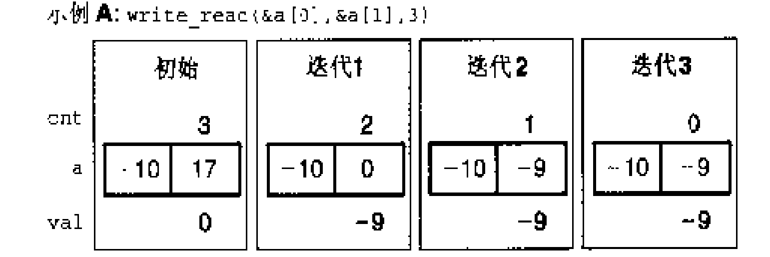
在图 5.33（a）的示例 B 中，参数 src 和 dest 都是指向数组元素 a[0]的指针。在此种情况中，指针引用*src 的每次加载都会得到指针引用*dest 的前次执行存储的值。因而，一系列不断增加的值会被存储在这个位置。通常，如果调用函数 write_read 时参数 src 和 dest 指向同一个存储器位置，而参数 cnt 的值为 $n>0$，那么净效果是将这个位置设置为 n-1。这个示例说明了一个现象，我们称之为写/读相关（write/read dependency）——一个存储器读的结果依赖于一个非常近的存储器写。我们的性能测试表明示例 B 的 CPE 为 6.00。写/读相关导致处理速度的下降。

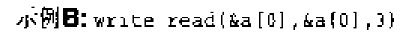
code/opt/copy.c
```
1   /* Write to dest, read from src */
2   void write_read(int *src, int *dest, int n)
```

```
3   {
4       int cnt = n;
5       int val = 0;
6
7       while (cnt--) {
8           *dest = val;
9           val = (*src)+1;
10      }
11  }
```

code/opt/copy.c

示例A: write_read(&a[0],&a[1],3)





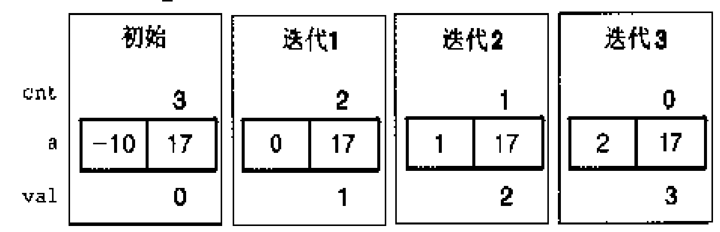


图 5.33 写和读存储器位置的代码以及示例执行

这个函数强调的是当参数 src 和 dest 相等时，存储和加载之间的相互影响。

为了了解处理器是如何区别这两种情况的，以及为什么一种情况比另一种运行得慢，我们必须更加仔细地看看加载和存储执行单元，如图 5.34 所示。存储单元包含一个存储缓冲区（store buffer），它包含已经被发射到存储单元而又还没有完成存储操作的地址和数据，这里的完成包括更新数据高速缓存。提供这样一个缓冲区，使得一系列存储操作不必等待每个操作更新高速缓存就能够执行。当一条加载操作发生时，它必须检查存储缓冲区中的条目，看有没有地址相匹配。如果有地址相匹配，它就取出相应的数据条目作为加载操作的结果。

其中内循环的汇编代码和它的第一次迭代到操作的翻译如下所示：

汇编指令	执行单元操作
.L32:	
movl %edx,(%ecx)	storeaddr (%ecx)
	storedata %edx.0
movl (%ebx),%edx	load (%ebx) → %edx.1a
incl %edx	incl %edx.1a → %edx.1b
decl %eax	decl %eax.0 → %eax.1
jnc .L32	jnc-taken cc.1

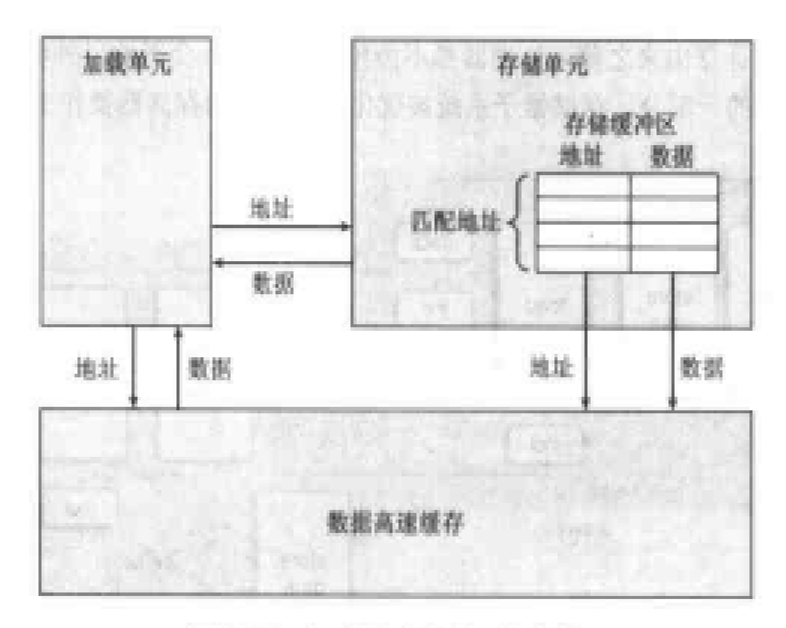


图 5.34 加载和存储单元的细节

存储单元包含一个未执行的写的缓冲区。加载单元必须检查它的地址是否与存储单元中的地址相符，以发现写/读相关。

我们看到，这里 `movl %edx,(%ecx)` 被翻译成两个操作：storeaddr 指令计算存储操作的地址，创建一个存储缓冲区中的条目，并设置该条目的地址字段；storedata 指令设置该条目的数据字段。因为只有一个存储单元，而存储操作是按照程序顺序处理的，所以两个操作如何协调是没有歧义的。正如我们看到的那样，这两个计算的执行是相互独立的这一事实对程序性能来说可能很重要。

图 5.35 给出了对于示例 A 的情况，write_read 的头两次迭代操作的时序。如 storeaddr 和 load 操作之间的虚线表明的那样，storeaddr 操作创建一个存储缓冲区中的条目，然后 load 会检查这个条目。因为这两个操作并不等价，所以 load 操作会继续从高速缓存中读数据。虽然存储操作还没有完成，处理器仍然能发现它影响的存储器位置不同于加载正试图读的位置。第二次迭代仍会重复这个过程。这里，我们能看到 storedata 操作必须等待，直到加载和增加了前一次迭代的结果。在此之前很久，storeaddr 操作和 load 操作可以比较它们的地址是否相同，确定它们是不同的，就允许加载操作继续进行。在我们的计算图中，展示了第二次迭代的加载就在第一次迭代的加载后 1 个周期开始执行。如果继续更多的迭代，我们就会发现这个图表明 CPE 为 1.0。很明显，某个其他的资源约束将实际性能限制在了 CPE 2.0 上。

图 5.36 给出了对于示例 B 的情况，write_read 的头两次迭代操作的时序。同样地，storeaddr 和 load 操作之间的虚线表明，storeaddr 操作创建一个存储缓冲区中的条目，然后 load 会检查这个条目。因为这些条目都是一样的，所以加载必须等待，直到 storedata 操作完成，然后它再从存储缓冲区中获得数据。这个等待在图中是以 load 操作加长了的方框来表示的。此外，我们展示了一条从 storedata 到 load 操作的虚线箭头，它表明 storedata 的结果被传递到 load 作为它的结果。我们这些操作的时序反映了测量出的 CPE 为 6.0。不过，这样的时序确切地是如何出现的，还不是完全清楚，所以这些图只是示意说明性的，而不是实际的。通常，处理器/存储器接口是处理器设计中最复杂的部分之一。不查阅详细的文档和使用机器分析工具，我们只能给出实际行为的一个假想的描述。

如这两个例子所示，存储器操作的实现包含很多细微的问题。对于寄存器操作，在指令解码成操作时，处理器就可以确定哪些指令会影响另外哪些指令。另一方面，对于存储器（或内存）操作，

在加载和存储地址被计算出来之前，处理器都不能预测哪些指令会影响另外哪些指令。由于存储器操作占到了程序很大的一部分，存储器子系统被优化成以独立的存储器操作来提供更大的并行性。

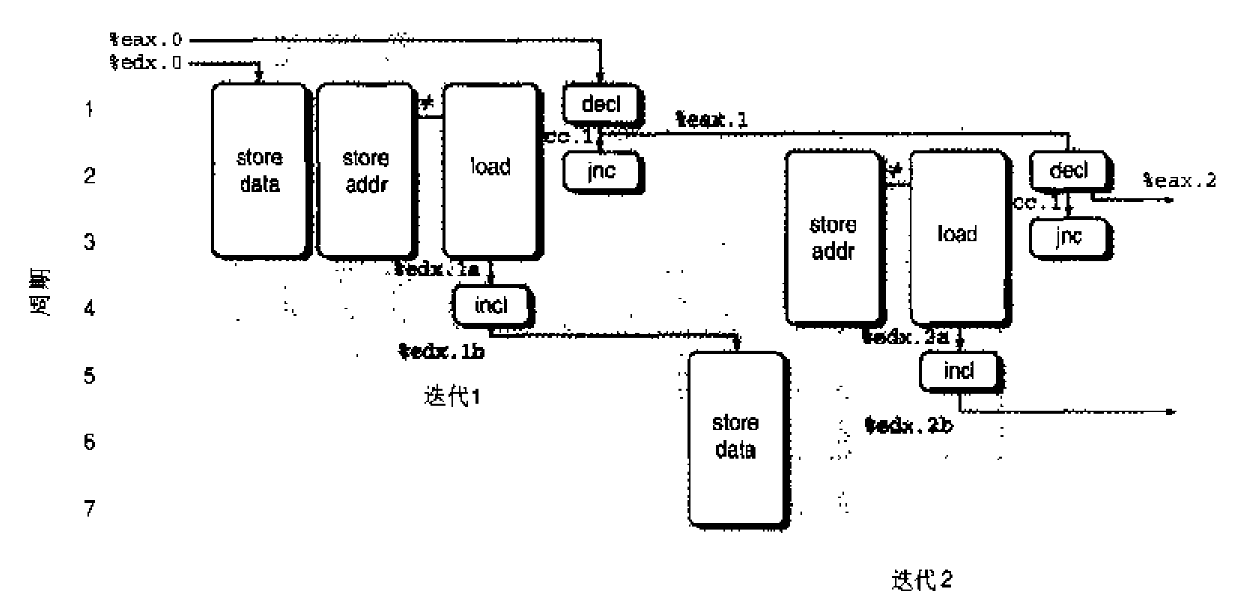


图 5.35 对示例 A 的 write_read 的时序

存储和加载操作有不同的地址，因此可以不等待存储就进行加载。

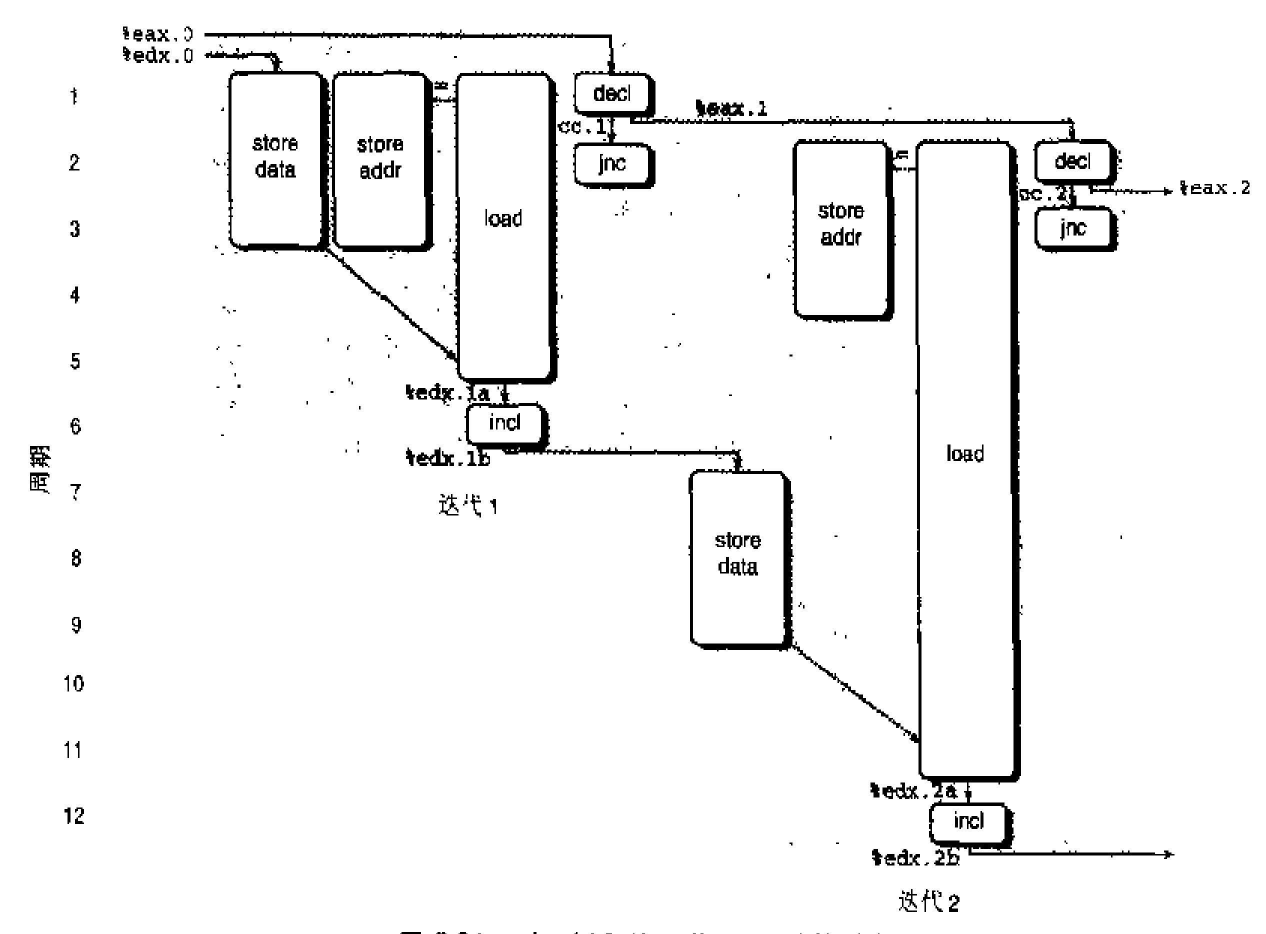


图 5.36 对示例 B 的 write_read 的时序

存储和加载操作有相同的地址，因此加载必须等待，直到它可以从存储获得结果了。

练习题 5.8

作为另一个具有潜在的加载-存储相互影响的代码，考虑下面的函数，它将一个数组的内容拷贝到另一个数组：

```
1    void copy_array(int *src, int *dest, int n)
2    {
3        int i;
4
5        for (i = 0; i < n; i++)
6            dest[i] = src[i];
7    }
```

假设 a 是一个长度为 1000 的数组，它被初始化为每个元素 a[i]等于 *i*。

A. 调用 copy_array(a+1, a, 999)的效果是什么？

B. 调用 copy_array(a, a+1, 999)的效果是什么？

C. 我们的性能测试表明问题 A 调用的 CPE 为 3.00，而问题 B 调用的 CPE 为 5.00。你认为是什么因素造成了这样的性能差异？

D. 调用 copy_array(a, a, 999)的性能会是怎样的？

5.14 现实生活：性能提高技术

虽然我们只考虑了有限的一组应用程序，但是我们能得出关于如何编写高效代码的很重要的经验教训。我们已经描述了许多优化程序性能的基本策略：

1. 高级设计。为手边的问题选择适当的算法和数据结构。要特别警觉，避免使用会渐进地产生糟糕性能的算法或编码技术。

2. 基本编码原则。避免限制优化的因素，这样编译器就能产生高效的代码。

- 消除连续的函数调用。在可能时，将计算移到循环外。考虑有选择的妥协程序的模块性以获得更大的效率。
- 消除不必要的存储器引用。引入临时变量来保存中间结果。只有在最后的值计算出来时，才将结果存放到数组或全局变量中。

3. 低级优化。

- 尝试各种与数组代码相对的指针形式。
- 通过展开循环降低循环开销。
- 通过诸如迭代分割之类的技术，找到使用流水线化的功能单元的方法。

要给读者最后的忠告是，要小心避免花费精力在令人误解的结果上。一项有用的技术是，在优化代码时使用检查代码（checking code）来测试代码的每个版本，以确保在这一过程中没有引入错误。检查代码将一系列测试应用到程序上，确保它得到期望的结果。当引入新的变量，改变循环边界，以及使代码整体更复杂时，很容易出错。此外，注意到性能上任何不同寻常的或出乎意料的变化是很重要的。正如我们已经表明的那样，由于性能异常，基准数据的选择能够在性能比较中造成很大的差异，因为我们只执行短指令序列。

5.15 确认和消除性能瓶颈

到此刻为止，我们只考虑了优化小的程序，在这样的小程序中，需要优化的地方很清楚。在处理大程序时，甚至于很难知道应该优化什么。在本节中，我们会描述如何使用代码剖析程序（code profilers），这是在程序执行时收集性能数据的分析工具。我们还展示了一个系统优化的通用原则，称为 Amdahl 定律（Amdahl's law）。

5.15.1 程序剖析

程序剖析（profiling）包括运行程序的这样一个版本，其中插入了工具代码，以确定程序的各个部分需要多少时间。确认出程序中我们需要集中注意力优化的部分是很有用的。剖析的一个有力之处在于可以一边在现实的基准数据（benchmark data）上运行实际的程序，一边进行剖析。

Unix 系统提供了一个剖析程序 GPROF。这个程序产生两种形式的信息。首先，它确定程序中每个函数花费了多少 CPU 时间。其次，它计算每个函数被调用的次数，以调用函数来分类。这两种形式的信息都非常有用。这些计时给出了不同函数在确定整体运行时间中的相对重要性。调用信息使得我们能理解程序的动态行为。

用 GPROF 进行剖析需要三个步骤，就像所示的对 C 程序 prog.c 那样，它运行时命令行参数为 file.txt：

1. 程序必须为剖析而编译和链接。使用 GCC（以及其他 C 编译器），就是在命令行上简单地包括运行时标志“-pg”。

```
unix> gcc -O2 -pg prog.c -o prog
```

2. 然后程序像往常一样执行：

```
unix> ./prog file.txt
```

它运行得会比正常时稍微慢一点，不过惟一的区别就是它产生了一个文件 gmon.out。

3. 调用 GPROF 来分析 gmon.out 中的数据。

```
unix> gprof prog
```

剖析报告的第一部分列出了执行各个函数花费的时间，按照降序排列。作为一个示例，下面列出了报告中关于程序中头三个函数的那一部分：

```
  %     cumulative    self              self      total
 time     seconds    seconds    calls  ms/call   ms/call  name
 85.62      7.80      7.80         1   7800.00   7800.00  sort_words
  6.59      8.40      0.60    946596      0.00      0.00  find_ele_rec
  4.50      8.81      0.41    946596      0.00      0.00  lower1
```

每一行代表对某个函数的所有调用所花费的时间。第一列表明花费在这个函数上的时间占整个时间的百分比。第二列显示的是直到这一行并包括这一行的函数所花费的累计时间。第三列显示的是花费在这个函数上的时间，而第四列显示的是它被调用的次数（递归调用不计算在内）。在我们的例子中，函数 sort_words 只被调用了一次，但就是这一次调用需要 7.80 秒，而函数 lower1 被调用了 946 596 次，总共需要 0.41 秒。

剖析报告的第二部分给出的是这个函数的调用历史。下面是一个递归函数 find_ele_rec 的历史：

```
                                4872758          find_ele_rec [5]
                  0.60    0.01  946596/946596    insert_string [4]
[5]      6.7      0.60    0.01  946596+4872758 find_ele_rec [5]
                  0.00    0.01  26946/26946      save_string [9]
                  0.00    0.00  26946/26946      new_ele [11]
                                4872758          find_ele_rec [5]
```

这个历史既显示了调用 find_ele_rec 的函数，也显示了它调用的函数。在上面一部分中，我们发现这个函数实际上被调用了 5 819 354 次(显示为"946596+4872758")——它自己调用了 4 872 758 次，而函数 insert_string（它本身被调用了 946 596 次）调用了 946 596 次。函数 find_ele_rec 依次调用了另外两个函数 save_string 和 new_ele，每个函数总共被调用了 26 946 次。

根据这个调用信息，我们通常可以推断出关于程序行为的有用信息。例如，函数 find_ele_rec 是一个递归过程，它扫描一个链表，查找一个特殊的字符串。假设递归与顶层调用的比率是 5.15，我们可以推断出它每次平均大约需要扫描 6 个元素。

GPROF 有些属性值得注意：

- 计时不是很准确。计时是基于一个简单的间隔计数（interval counting）机制的，在第 9 章中会讨论这个问题。简而言之，编译过的程序为每个函数维护一个计数器，记录花费在执行该函数上的时间。操作系统使得每隔某个规则的时间间隔 δ，程序被中断一次。δ 的典型值的范围为 1.0～10.0 毫秒。当中断发生时，它会确定程序正在执行什么函数，并将该函数的计数器值增加 δ。当然，也可能这个函数只是刚开始执行，而很快就会完成，却赋给它从上次中断以来整个的执行花费。在两次中断中也可能运行其他某个程序，却因此根本没有计算花费。

对于运行时间较长的程序，这种机制工作得相当好。从统计上来说，应该根据花费在执行函数上的相对时间来对每个函数计算花费。不过，对于那些运行时间少于 1 秒的程序来说，得到的统计数字只能看成是粗略的估计值。

- 调用信息相当可靠。编译过的程序为每对调用者和被调用者维护一个计数器。每次调用一个过程时，就会对适当的计数器加 1。
- 默认情况下，不会显示对库函数的调用。作为替代，对库函数调用的次数体现在了调用函数的次数中。

5.15.2 使用剖析程序来指导优化

作为一个用剖析程序来指导程序优化的示例，我们创建了一个包括几个不同任务和数据结构的程序。这个应用程序读一个文本文件，创建一张互不相同的单词和每个单词出现次数的表，然后按照出现次数的降序对单词排序。作为基准程序，我们在一个由莎士比亚全集组成的文件上运行这个程序。据此，我们确定莎士比亚一共写了 946 596 个单词，其中 26 946 是互不相同的。最常见的单词是"the"，出现了 29 801 次。单词"love"出现了 2249 次，而"death"出现了 933 次。

我们的程序是由下列部分组成的。我们创建了一系列的版本，从各部分简单的算法开始，然后再换成更成熟完善的算法：

1. 从文件中读出每个单词，并转换成小写字母。我们最初的版本使用的是函数 lower1（图 5.7），我们知道它的复杂度是二次的。

2. 对字符串应用一个哈希函数，为一个有 s 个表元（buckets）的哈希表产生一个 0～s-1 之间的数字。我们最初的函数只是简单地对字符的 ASCII 代码求和，再对 s 求模。

3. 每个哈希表元都组织成一个链表。程序沿着这个链表扫描，寻找一个匹配的条目。如果找到了，该单词的频度就加 1。否则，就创建一个新的链表元素。我们最初的版本递归地完成这个操作，将新元素插在链表尾部。

4. 一旦已经生成了这张表，我们就根据频度对所有的元素排序。我们最初的版本使用插入排序。

图 5.37 给出了我们的单词频度分析程序各个版本的剖析结果。对于每个版本，我们将时间分为五类。

Sort：按照频度对单词排序。

List：为匹配单词扫描链表，如果需要，插入一个新的元素。

Lower：将字符串转换为小写字母。

Hash：计算哈希函数。

Rest：其他所有函数的和。

如图中（a）部分所示，我们最初的版本需要 9 秒多钟，大多数时间花在了排序上。这并不奇怪，因为插入排序有二次复杂度，而程序对 27 000 个值进行排序。

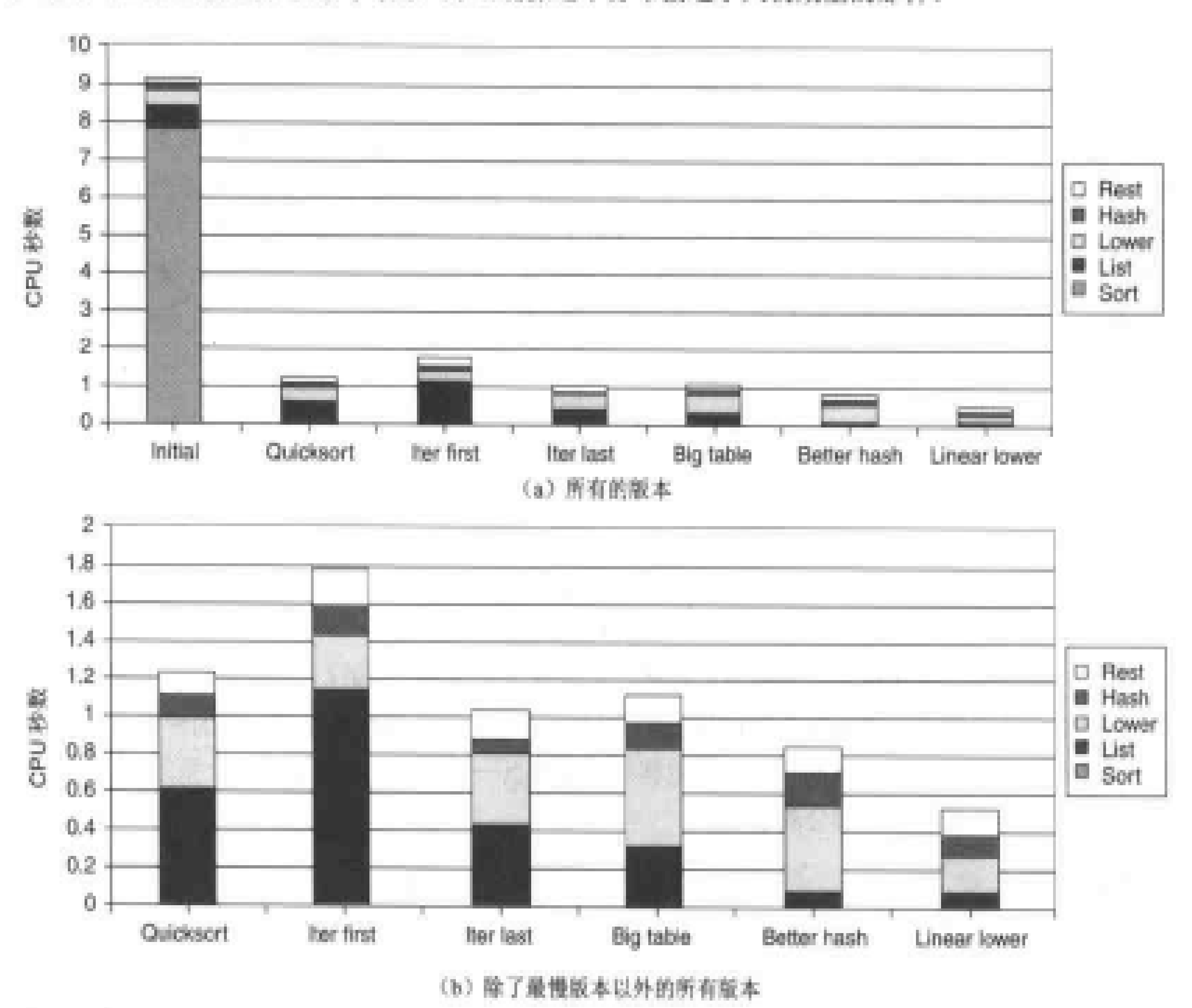
在我们下一个版本中，我们用库函数 qsort 进行排序，这个函数是基于快速排序算法的。在图中这个版本称为“Quicksort”。更有效的排序算法使花在排序上的时间降低到可以忽略不计，而整个运行时间降低到大约 1.2 秒。图的（b）部分给出的是剩下各个版本的时间，所用的比例能使我们看得更清楚。

改进了排序，现在我们发现链表扫描变成了瓶颈。想想这个低效率是由于函数的递归结构引起的，我们用一个迭代的结构替换它，显示为“Iter First”。令人奇怪的是，运行时间增加到了大约 1.8 秒。根据更近一步的研究，我们发现两个链表函数之间有一个细微的差别。递归版本将新元素插入到链表尾部，而迭代版本把它们插到链表头部。为了使性能最大化，我们希望频率最高的单词出现在链表的开始处。这样一来，函数就能快速地定位常见的情况。假设单词在文档中是均匀分布的，我们期望频度高的单词的第一次出现在频度低的单词的第一次出现之前。通过将新单词插入尾部，第一个函数倾向于按照频度的降序排序，而第二个函数则相反。因此我们创建第三个使用迭代的链表扫描函数，不过是将新元素插入到链表的尾部。使用这个版本，显示为“Iter Last”，时间降到了大约 1.0 秒，比递归版本稍微好一点。

接下来，我们考虑哈希表的结构。最初的版本只有 1021 个表元（通常，会选择表元的个数为素数，以增强哈希函数将关键字均匀分布在表元中的能力）。对于一个有 26 946 个条目的表来说，这就意味着平均负载（load）是 26 946/1007=26.4。这就解释了为什么有那么多时间花在了执行链表操作上了——搜索包括测试大量的候选单词。它还解释了为什么性能对链表顺序这么敏感了。因而，我们将表元的数量增加到了 10 007，将平均负载降低到了 2.70。不过，很奇怪的是，我们的整体运行时间增加到了 1.11 秒。剖析结果表明增加的时间主要花在了小写字母转换函数上，虽然不大可能是这样的。我们的运行时间过于短了，我们不能期望这些计时非常精确。

我们假设表变大了而性能变差了是因为哈希函数选择得不太好。简单地对字符编码求和不能产生一个大范围的值，也不能根据字符的分类做出区分。例如，单词“god”和“dog”都会哈希到位置 147+157+144=448，因为它们包含相同的字符。单词“foe”也会哈希到这个位置，因为

146+157+145=448。我们换成一个使用移位和异或操作的哈希函数。使用这个版本，显示为“Better Hash”，时间下降到了 0.84 秒。一个更加系统化的方法是更加仔细地研究关键字在表元中的分布，如果哈希函数的输出分布是均匀的，那么确保这个分布接近于人们期望的那样。



(a) 所有的版本

(b) 除了最慢版本以外的所有版本

图 5.37　单词频度计数程序各个版本的剖析结果时间是根据程序中不同的主要操作划分的

最后，我们把运行时间降到了一半的时间是花在执行小写字母转换上了。我们已经看到了函数 lower1 的性能很差，特别是对长字符串来说。这篇文档中的单词足够短，能避免二次性能（quadratic performance）的灾难性的结果：最长的单词（“honorificabilitudinitatibus”）长度为 27 个字符。不过换成使用 lower2，显示为“Linear Lower”得到很好的性能，整个时间降到了 0.52 秒。

通过这个练习，我们展示了代码剖析能够帮助将一个简单应用程序所需的时间从 9.11 秒降低到 0.52 秒——提高到了 17.5 倍。剖析程序帮助我们把注意力集中在程序最耗时的部分上，同时还提供了关于过程调用结构的有用信息。

我们可以看到，剖析是工具箱中一个很有用的工具，但是它不应该是惟一一个。计时测量不是很准确，特别是对较短的运行时间（小于 1 秒）来说。结果只适用于被测试的那些特殊的数据。例如，如果我们在由较少数量的较长字符串组成的数据上运行最初的函数，我们会发现小写字母转换函数才是主要的性能瓶颈。更糟糕的是，如果它只剖析包含短单词的文档，我们可能不会发现隐藏

着的性能杀手，例如 lower1 的二次性能。通常，假设我们在有代表性的数据上运行程序，剖析能帮助我们对典型的情况进行优化，但是我们还应该确保对所有可能的情况，程序都有相当的性能。这主要包括避免得到糟糕的渐近性能（asymptotic performance）的算法（例如插入算法）和坏的编程实践（例如 lower1）。

5.15.3 Amdahl 定律

Gene Amdahl（计算领域的先驱之一）做出了一个关于提高系统一部分性能的效果的简单但是富有洞察力的观察。这个观察现在被称为 Amdahl 定律。其主要思想是当我们加快系统一个部分的速度时，对系统整体性能的影响依赖于这个部分有多重要和速度提高了多少。考虑一个系统，在其中执行某个应用程序需要时间 T_{old}。假设系统的某个部分需要这个时间的百分比为 α，而我们将它的性能提高到了 k 倍。也就是，这个部分原来需要时间 αT_{old}，而现在需要时间$(\alpha T_{old})/k$。因此，整个执行时间会是

$$\begin{aligned} T_{new} &= (1-\alpha)T_{old} + (\alpha T_{old})/k \\ &= T_{old}[(1-\alpha) + \alpha/k] \end{aligned}$$

据此，我们可以计算加速 $S = T_{old}/T_{new}$ 为：

$$S = \frac{1}{(1-\alpha)+\alpha/k} \tag{5.1}$$

作为一个示例，考虑这样一种情况，系统原来占用 60%时间（α=0.6）的部分被提高到了 3 倍（k=3）。那么我们得到加速 1/[0.4+0.6/3]=1.67。因此，即使我们大幅度改进了系统的一个主要部分，我们的净加速还是很小。这就是 Amdahl 定律的主要观点——要想大幅度提高整个系统的速度，我们必须提高整个系统很大一部分的速度。

练习题 5.9

假设你的职业是卡车司机，你被雇佣运送一车土豆从 Idaho 的 Boise 到 Minnesota 的 Minneapolis，总距离为 2500 公里，你估计在速度限制以内你开车的平均时速为 100 公里，整个行程需要 25 小时。

A. 你在新闻里听说 Montana 刚刚取消了它的限速，这段路程有 1500 公里。你的卡车可以开到每小时 150 公里。你这次行程的加速（speedup）会是多少？

B. 你可以在 www.fasttrucks.com 为你的卡车购买一个新的涡轮增压器。它们有许多样式，不过想开得越快，花费就越大。要想行程加速达到 5/3，你必须以多大的速度通过 Montana?

练习题 5.10

你公司的市场部门许诺你的客户下一版软件性能会提高一倍。分配给你的任务是就这个承诺发表意见。你确定只能改进系统 80%的部分。为了达到整体性能目标，你需要将这个部分提高到多少（也就是，k 的值应为多少）？

Amdahl 定律的一个有趣的特殊情况是考虑将 k 设为∞的效果。也就是，我们能够取出系统的某个部分，把它的速度提高到时间可以忽略不计的程度。那么我们得到

$$S_{\infty} = \frac{1}{(1-\alpha)} \tag{5.2}$$

因此，例如，如果我们能够将系统60%的部分速度提高到它所需要的时间接近于0，那么我们的净增速也仍然只为1/0.4=2.5。当我们用快速排序取代插入排序时，从我们的字典程序中就能看出这个性能。最开始的版本花费它9.1秒中的7.8秒来进行插入排序，得到α=0.86。使用快速排序，花费在排序上的时间变得可以忽略不计，得到预测的增速为7.1。实际上，实际的增速要高一点：9.11/1.22=7.5，这是由于对初始版本的剖析测试的不准确性造成的。我们能够获得大的增速，这是因为排序占到了整个执行时间的一个非常大的比例。

Amdahl定律描述了一个改进任何过程的通用原则。除了适用于提高计算机系统的速度之外，它还能指导一个公司试着降低生产剃须刀的成本，或是指导一个学生改进他或她的平均绩点。或许它在计算机世界里最有意义，在计算机世界中，我们通常将性能提高一倍或更多。只有通过优化系统很大的一部分才能获得这么高的提高率。

5.16 小结

虽然关于代码优化的大多数论述都描述了编译器是如何能生成高效代码的，但是应用程序员有很多方法来协助编译器完成这项任务。没有任何编译器能用一个好的算法或数据结构代替低效率的算法或数据结构，因此程序设计的这些方面仍然应该是程序员主要关心的。我们还看到妨碍优化的因素，例如存储器别名和过程调用，严重限制了编译器执行大量优化的能力。同样，程序员必须对消除这些妨碍优化的因素负主要的责任。

除此之外，我们还研究了一系列技术，包括循环展开、迭代分割以及指针运算。随着我们对优化的深入，研究汇编代码以及试着理解机器是如何执行计算的变得重要起来。对于现代、乱序处理器上的执行，分析程序是如何在有无限处理资源但是功能单元的执行时间和发射时间与目标处理器相符的机器上执行的，收获良多。为了精练这个分析，我们还应该考虑诸如功能单元数量和类型这样的资源约束。

包含条件分支或与存储器系统复杂交互的程序，比我们首先考虑的简单循环程序，更加难以分析和优化。基本策略是使循环更容易预测，并试着减少存储和加载操作之间的相互影响。

当处理大型程序时，将我们的注意力集中在最耗时的部分变得很重要。代码剖析程序和相关的工具能帮助我们系统地评价和改进程序性能。我们描述了GPROF，一个标准的Unix剖析工具。也还有更加复杂完善的剖析程序可用，例如Intel的VTUNE程序开发系统。这些工具可以在过程级分解执行时间，测量程序每个基本块（basic block）的性能。基本块是没有条件操作的指令序列。

Amdahl定律提供了对通过只改进系统一部分所获得的性能收益的一个简单但是很有力的看法。收益既依赖于我们对这个部分的提高程度，也依赖于这个部分原来在整个时间中所占的比例。

参考文献说明

有许多关于编译器优化技术的作品。Muchnick的著作被认为是最全面的[55]。Wadleigh和Crawford的关于软件优化的著作[85]包含了一些我们已经谈到的内容，不过它还描述了在并行机器上获得高性能的过程。

我们对乱序处理器的操作的描述相当简单和抽象。可以在高级计算机体系结构教科书中找到对通用原则更完整的描述，例如Hennessy和Patterson的著作[33，第3章]。Shriver和Smith给出了

AMD处理器的详细描述[69]，AMD处理器与我们描述的处理器有相似之处。

大多数关于计算机体系结构的书都讲述了Amdahl定律。Hennessy和Patterson的著作[33]主要关心的是量化的系统评价，并提供了对这个主题相当好的讲解。

家庭作业

5.11 ◆◆

假设我们想编写一个计算两个向量内积的过程。这个函数的一个抽象版本对整数和浮点数据都有CPE 54，通过进行与我们将抽象combine1变换为更有效的combine4相同类型的变换，我们得到如下代码：

```
1   /* Accumulate in temporary */
2   void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3   {
4       int i;
5       int length = vec_length(u);
6       data_t *udata = get_vec_start(u);
7       data_t *vdata = get_vec_start(v);
8       data_t sum = (data_t) 0;
9
10      for (i = 0; i < length; i++) {
11          sum = sum + udata[i] * vdata[i];
12      }
13      *dest = sum;
14  }
```

我们的测试显示对于整数数据，这个函数每次迭代需要3.11个周期。其中循环的汇编代码如下所示：

```
     udata in %esi, vdata in %ebx, i in %edx, sum in %ecx, length in %edi
1   .L24:                                   loop:
2     movl (%esi,%edx,4),%eax                 Get udata[i]
3     imull (%ebx,%edx,4),%eax                Multiply by vdata[i]
4     addl %eax,%ecx                          Add to sum
5     incl %edx                               i++
6     cmpl %edi,%edx                          Compare i:length
7     jl .L24                                 if <, goto loop
```

假设整数乘法是由通用整数功能单元执行的，而这个单元是流水线化的。这意味着在一个乘法开始之后一个周期，一个新的整数操作（乘法或其他操作）就能开始了。还假设整数/分支功能单元能执行简单的整数操作。

A. 给出这些汇编代码行到操作序列的翻译。movl指令翻译成一条load操作。寄存器%eax在循环中被更新两次。用标号区分不同版本的%eax.1a和%eax.1b。

B. 解释函数怎么能比整数乘法需要的周期数运行得还快。

C. 解释是什么因素限制了这段代码的CPE最好也只能为2.5。

D. 对于浮点数据，我们得到的CPE为3.5。不需要检查汇编代码，描述将性能限制在最好情况

为每次迭代3个周期的一个因素。

5.12 ◆

编写习题5.11中描述的一个版本的内积过程，使用四次循环展开。

我们对这个过程的测试得到对整数数据CPE为2.20，而对浮点数据CPE为3.50。

A. 解释为什么任何版本的内积过程都不能达到比2更大的CPE了。

B. 解释为什么对浮点数的性能不能通过循环展开而得到提高。

5.13 ◆

编写习题5.11中描述的一个版本的内积过程，使用四次循环展开和两路并行。

我们对这个过程的测试得到对浮点数的CPE为2.25。描述将性能限制在最好CPE为2.0的两个因素。

5.14 ◆◆

你刚刚加入一个编程小组，他们试图开发世界上最快的阶乘函数。开始时使用递归阶乘，他们将代码转换成使用迭代：

```
1   int fact(int n)
2   {
3       int i;
4       int result = 1;
5
6       for (i = n; i > 0; i--)
7           result = result * i;
8       return result;
9   }
```

通过这样做，他们将函数的CPE数从63降低到4，这是在Intel Pentium III上测出的（真的!）。不过，他们还想干得再好一点。

其中一个程序员听说过循环展开，她写出了如下代码：

```
1   int fact_u2(int n)
2   {
3       int i;
4       int result = 1;
5       for (i = n; i > 0; i-=2) {
6           result = (result * i) * (i-1);
7       }
8       return result;
9   }
```

不幸的是，小组发现这段代码对参数n的某些值返回0。

A. 对于哪些n值，fact_u2和fact会返回不同的值？

B. 给出如何修正fact_u2。注意，对于这个过程有个特殊的窍门，只要修改一个循环界限。

C. 对fact_u2使用基准程序，显示性能没有改进。你会如何解释这个现象呢？

D. 你将循环中的这行修改为

```
6              result = result * (i * (i - 1));
```

让每个人惊奇的是，现在测出的性能有 CPE 2.5。你怎样解释这个性能改进呢？

5.15 ◆

使用条件传送指令，编写下列函数体的汇编代码：

```
1   /* Return maximum of x and y */
2   int max(int x, int y)
3   {
4       return (x < y) ? y : x;
5   }
```

5.16 ◆◆

使用条件传送，翻译如下形式的语句

val = *cond-expr* ? *then-expr* : *else-expr*;

的通用技术产生了如下形式的代码

```
val = then-expr;
temp = else-expr;
test = cond-expr;
if (test) val = temp;
```

这里最后一行是用一个条件传送指令来实现的。以练习题 5.7 为例，说明这个翻译合法的通用要求。

5.17 ◆◆

下面这个函数计算的是一个链表中元素的和：

```
1   int list_sum(list_ptr ls)
2   {
3       int sum = 0;
4
5       for (; ls; ls = ls->next)
6           sum += ls->data;
7       return sum;
8   }
```

循环的汇编代码和第一次迭代到操作的翻译如下：

汇编指令	执行单元操作		
.L43:			
addl 4(%edx),%eax	movl 4(%edx.0)	→	t.1
	addl t.1,%eax.0	→	%eax.1
movl (%edx),%edx	load (%edx.0)	→	%edx.1
testl %edx,%edx	testl %edx.1,%edx.1	→	cc.1
jne .L43	jne-taken cc.1		

A. 按照图 5.31 的风格，画图说明循环头三次迭代的操作的调度。回想一下只有一个加载单元。

B. 我们对这个函数的测试得到 CPE 为 4.00。这与你在 A 部分中画出的图一致吗？

5.18 ◆◆

下面的函数是问题 5.17 中所示的链表求和函数的一个变种：

```
1   int list_sum2(list_ptr ls)
2   {
3       int sum = 0;
4       list_ptr old;
5
6       while (ls) {
7          old = ls;
8          ls = ls->next;
9          sum += old->data;
10      }
11      return sum;
12  }
```

这段代码的编写方式，使得取下一个链表元素的存储器访问早于从当前元素取数据字段的存储器访问。

循环的汇编代码和第一次迭代到操作的翻译如下：

汇编指令	执行单元操作		
.L48:			
movl %edx,%ecx			
movl (%edx),%edx	load (%edx.0)	→	%edx.1
addl 4(%ecx),%eax	movl 4(%edx.0)	→	t.1
	addl t.1,%eax.0	→	%eax.1
testl %edx,%edx	testl %edx.1,%edx.1	→	cc.1
jne .L48	jne-taken cc.1		

注意，寄存器传送操作 `movl %edx, %ecx` 不需要用任何操作来实现。它的处理只要简单地将标记 edx.0 与寄存器%ecx 联系起来，这样一来，后面的指令 `addl 4(%ecx), %eax` 就会被翻译成以 edx.0 作为它的源操作数。

A. 按照图 5.31 的风格，画图说明循环头三次迭代的操作的调度。回想一下只有一个加载单元。

B. 我们对这个函数的测试得到 CPE 为 3.00。这与你在 A 部分中画出的图一致吗？

C. 这个函数比问题 5.17 中的函数怎样更好地利用了加载单元？

5.19 ◆

假设给了你一个任务，要提高一个由 3 个部分组成的程序的性能。部分 A 需要整个运行时间的 20%，部分 B 需要 30%，而部分 C 需要 50%。你确定 1000 美元能将部分 B 的速度提高到 3.0 倍，也可以将部分 C 的速度提高到 1.5 倍。哪种选择会使性能最大化？

练习题答案

练习题 5.1 答案

这个问题说明了存储器别名的某些细微的影响。

正如下面加了注释的代码所示，结果会是将 xp 处的值设置为 0:

```
1   *xp = *xp + *xp; /* 2x */
2   *xp = *xp - *xp; /* 2x-2x = 0 */
3   *xp = *xp - *xp; /* 0-0 = 0 */
```

这个示例说明我们关于程序行为的直觉往往会是错误的。我们自然地会认为 xp 和 yp 是不同的情况，却忽略了他们相等的可能性。错误通常源自程序员没想到的情况。

练习题 5.2 答案

这个问题说明了 CPE 和绝对性能之间的关系。可以用初等代数解决这个问题。我们发现对于 $n \leq 2$，版本 1 最快。对于 $3 \leq n \leq 2$，版本 2 最快，而对于 $n \geq 8$，版本 3 最快。

练习题 5.3 答案

这是个简单的练习，但是认识到一个 for 循环的四个语句（初始化、测试、更新和循环体）执行的次数是不同的很重要。

代码	min	max	incr	square
A.	1	91	90	90
B.	91	1	90	90
C.	1	1	90	90

练习题 5.4 答案

正如我们在第 3 章中发现的，从汇编代码到 C 代码的逆向工程提供了对编译过程的有用见识。下面的代码给出了对于通用数据和通用合并操作的形式：

```
1    void combine5px8(vec_ptr v, data_t *dest)
2    {
3        int length = vec_length(v);
4        int limit = length - 3;
5        data_t *data = get_vec_start(v);
6        data_t x = IDENT;
7        int i;
8
9        /* Combine 8 elements at a time */
10       for (i = 0; i < limit; i+=8) {
11           x = x OPER data[0]
12               OPER data[1]
13               OPER data[2]
14               OPER data[3]
15               OPER data[4]
16               OPER data[5]
```

```
17                OPER data[6]
18                OPER data[7];
19            data += 8;
20        }
21
22        /* Finish any remaining elements */
23        for (; i < length; i++) {
24            x = x OPER data[0];
25            data++;
26        }
27        *dest = x;
28    }
```

我们手写的指针代码通过计算指针的结束值，能够消除循环变量 i。这又是一个训练有素的人常常能够看出那些被编译器忽略了的变换的示例。

练习题 5.5 答案

溢出的（spilled）值通常存储在本地栈帧中。因此，它们相对于%ebp 的偏移为负。我们可以在汇编代码中第 12 行上看到这样一个引用。

A. 变量 limit 被溢出到栈中。

B. 它在相对于%ebp 偏移为-8 处。

C. 只有在确定是否会选择结束循环的 jl 指令时才会需要这个值。如果分支预测逻辑预测会选择分支，那么下一次迭代就能在循环测试完成之前进行。因此，比较指令不是确定循环性能的关键路径的一部分。此外，因为这个变量不会在循环中被改变，所以把它放到栈中不需要任何额外的存储操作。

练习题 5.6 答案

这个问题证明了程序中很小的改动是如何能够造成巨大的性能差异的，特别是在乱序执行的机器上。图 5.38 表示了函数针对每种结合的一次迭代的乘法操作的调度。每次迭代包括三个乘法，而每个乘法接收 r 的旧值（显示为 r.0）并计算一个新的值（显示为 r.1）。不过，如灰色虚线所示，关键路径（critical path），也就是对 r 的连续更新之间的最小时间可以是 12（A1）、8（A2 和 A5）或 4（A3 和 A4）。假设处理器达到最大的并行度，那么只有这个关键路径会限制 CPE 的理论值。

这会得到下面的表：

版本	测量的 CPE	理论 CPE
A1	4.00	12/3 = 4.00
A2	2.67	8/3 = 2.67
A3	1.67	4/3 = 1.33
A4	1.67	4/3 = 1.33
A5	2.67	8/3 = 2.67

从这张表我们看出结合 A1、A2 和 A5 达到了它们的理论最优值，而 A2 和 A3 每次迭代要花费 5 个周期，而不是理论上的最优值 4。

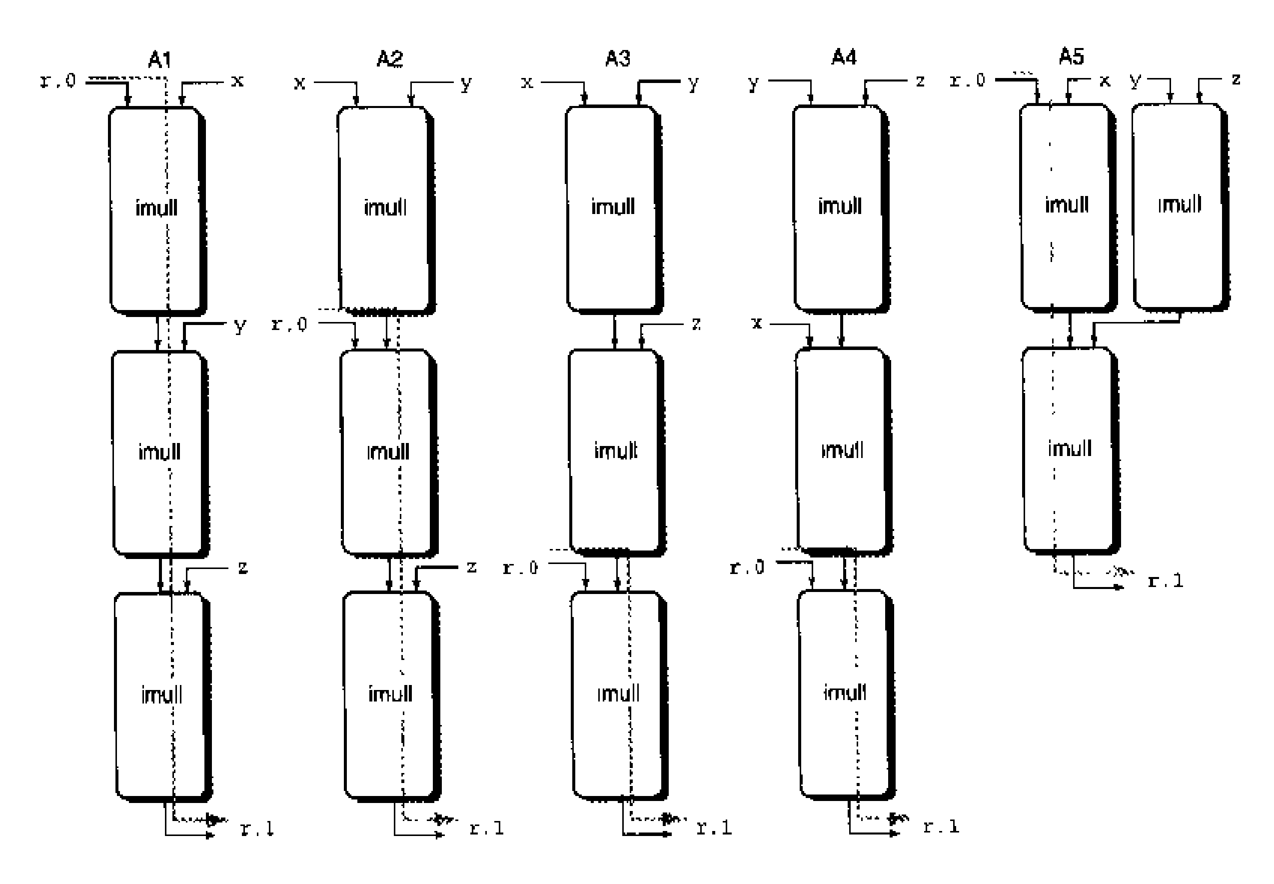


图 5.38 问题 5.6 中情况的乘法操作的调度

灰色虚线表示限制变量 r 的连续更新之间时间的关键路径。

练习题 5.7 答案

这个问题证明了当使用条件传送时需要小心。它们要求对源操作数求值，甚至于在不使用这个值时。

这段代码总是间接引用 xp（汇编代码的第 2 行）。在 xp 为 0 的情况中，这会导致一个空指针引用。

练习题 5.8 答案

这个问题要求你分析一个程序中潜在的 load-store 交互作用。

A. 它会将每个元素 a[i]设置为 i+1， $0 \le i \le 998$。

B. 它会将每个元素 a[i]设置为 0， $0 \le i \le 999$。

C. 在第二种情况中，一次迭代的加载取决于前次迭代存储的结果。因此，在连续的迭代之间有写/读相关。

D. 它会得到 CPE 5.00，因为存储和后续的加载之间没有相关。

练习题 5.9 答案

这个问题说明了 Amdahl 定律不仅仅只适用于计算机系统。

A. 按照等式 5.1，我们有 α= 0.6 和 k = 1.5。更直观地说，穿过 Montana 行驶 1500 公里需要 10 个小时，而剩下的行程也需要 10 个小时。这会得到增速 25/(10+10)=1.25。

B. 按照等式 5.1，我们有 α= 0.6，而我们需要 S=5/3，根据这些我们可以解出 k。更直观地说，

为了使行程加速5/3，我们必须将整个时间降低到15个小时。Montana之外的部分仍然需要10个小时，所以我们必须在5个小时内通过Montana。这要求行驶速度为每小时300公里，对于卡车来说实在是太快了！

练习题5.10答案

通过一些示例是理解Amdahl定律的最好方法。这个例子要求你从一个不同寻常的角度来看等式5.1。

这个问题是这个等式的一个简单应用。给定$S=2$和$\alpha=0.8$，而你必须解出k：

$$\begin{aligned} 2 &= \frac{1}{(1-0.8)+0.8/k} \\ 0.4+1.6/k &= 1.0 \\ k &= 2.67 \end{aligned}$$

CHAPTER 6

存储器层次结构

到目前为止，在我们对系统的研究中，我们依赖于一个简单的计算机系统模型，CPU执行指令，而存储器（memory）系统为CPU存放指令和数据。在我们简单的模型中，存储器系统是一个线性的字节数组，而CPU能够在一个常数时间内访问每个存储器位置。虽然迄今为止这都是一个有效的模型，但是它没有反映现代系统实际工作的方式。

实际上，存储器系统（memory system）是一个具有不同容量、成本和访问时间的存储（storage）设备的层次结构。CPU寄存器保存着最常用的数据。靠近CPU的小的、快速的高速缓存存储器（cache memory）作为存储（stored）在相对慢速的主存储器（main memory，简称主存）中数据和指令子集的缓冲区域。主存暂时存放存储在较大的慢速磁盘上的数据，而这些磁盘常常又作为存储在通过网络连接的其他机器的磁盘或磁带上的数据的缓冲区域。

存储器层次结构是可行的，这是因为与下一个更低层次的存储设备相比来说，一个编写良好的程序倾向于更频繁地访问某一个层次上的存储设备。所以，下一层的存储设备可以更慢速一点，也因此更大，每个位更便宜。整体效果是一个大的存储器池，其成本与层次结构底层最便宜的存储设备相当，但是却以接近于层次结构顶部存储设备的高速率向程序提供数据。

作为一个程序员，你需要理解存储器层次结构，因为它对你应用程序的性能有着巨大的影响。如果你的程序需要的数据是存储在CPU寄存器中的，那么在执行期间，在零个周期内就能访问到它们。如果存储在高速缓存中，需要1～10个周期。如果存储在主存中，需要50～100个周期。而如果存储在磁盘上，需要大约20 000 000个周期！

这里就是计算机系统中一个基本而持久的思想：如果你理解了系统是如何将数据在存储器层次结构中上上下下移动的，那么你可以编写你的应用程序，使得它们的数据项存储在层次结构中较高的地方，在那里CPU能更快地访问到它们。

这个思想围绕着计算机程序的一个称为局部性（locality）的基本属性。具有良好局部性的程序倾向于一次又一次地访问相同的数据项集合，或是倾向于访问邻近的数据项集合。具有良好局部性的程序比局部性差的程序更多地倾向于从存储器层次结构中较高层次处访问数据项，因此运行得更快。例如，不同的矩阵乘法核心程序执行相同数量的算术操作，但是有不同程度的局部性，它们的运行时间可以相差6倍！

在本章中，我们会看看基本的存储技术——SRAM存储器、DRAM存储器、ROM存储器和磁盘——并描述它们是如何被组织成层次结构的。特别地，我们将注意力集中在CPU和主存之间作为缓存区域的高速缓存存储器上，因为它们对应用程序性能的影响最大。我们向你展示如何分析你的C程序的局部性，而且我们还介绍改进你的程序中局部性的技术。你还会学到一种描绘某台机器上存储器层次结构的性能的有趣方法，称为“存储器山（memory mountain）”，它给出的读访问次数是局部性的一个函数。

6.1 存储技术

计算机技术的成功很大程度上源自于存储技术的巨大进步。早期的计算机只有几千字节的随机访问存储器。最早的IBM PC甚至于没有硬盘。1982年引入的IBM PC-XT有10M字节的磁盘。到2000年，主流机器已有1000倍于PC-XT的磁盘存储器，而且这个比率会以每两年或三年10倍的速度增长。

6.1.1 随机访问存储器

随机访问存储器（random-access memory，RAM）分为两类——静态的和动态的。静态 RAM（SRAM）比动态 RAM（DRAM）更快，但也贵得多。SRAM 用来作为高速缓存存储器，既可以在 CPU 芯片上，也可以不在 CPU 芯片上。DRAM 用来作为主存以及图形系统的帧缓冲区。典型地，一个桌面系统的 SRAM 不会超过几兆字节，但是 DRAM 却有几百或几千兆字节。

静态 RAM

SRAM 将每个位存储在一个双稳态的（bistable）存储器单元（cell）里。每个单元是用一个六晶体管电路来实现的。这个电路有这样一个属性，它可以无限期地保持在两个不同的电压配置（configuration）或状态（state）之一。其他任何状态都是不稳定的——从那里开始，电路会迅速地转移到两个稳定状态中的一个。这样一个存储器单元类似于图 6.1 中画出的倒转的钟摆。

当钟摆倾斜到最左边或最右边时，它是稳定的。从其他任何位置，钟摆都会倒向一边或另一边。原则上，钟摆也能在垂直的位置无限期地保持平衡，但是这个状态是亚稳态的（metastable）——最细微的扰动也能使它倒下，而且一旦倒下就永远不会再恢复到垂直的位置。

由于 SRAM 存储器单元的双稳态特性，只要有电，它就会永远地保持它的值。即使有干扰，例如电子噪音，来扰乱电压，当干扰消除时，电路就会恢复到稳定值。

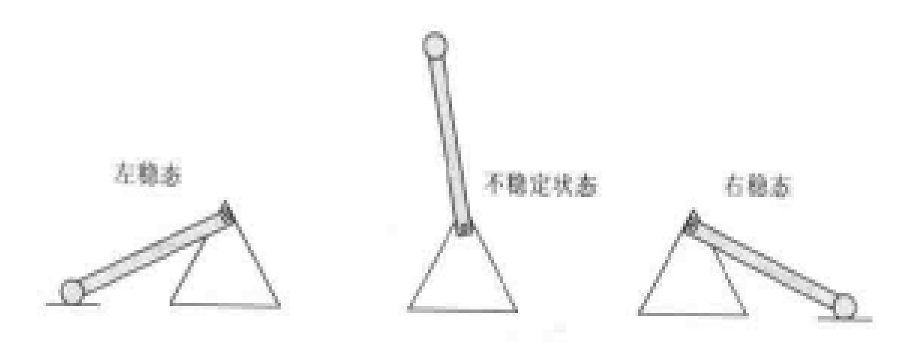


图 6.1 倒转的钟摆

同 SRAM 单元一样，钟摆只有两个稳定的组态或状态。

动态 RAM

DRAM 将每个位存储为对电容的充电。这个电容非常小，通常只有大约千万亿分之一法拉——也就是，30×10^{-15} 法拉。不过，回想一下法拉是一个非常大的计量单位。DRAM 存储器可以制造得非常密集——每个单元由一个电容和一个访问晶体管组成。但是，与 SRAM 不同，DRAM 存储器单元对干扰非常敏感。当电容的电压被扰乱之后，它就永远不会恢复了。暴露在光线下会导致电容电压改变。实际上，数码照相机和摄像机中的传感器本质上就是 DRAM 单元的阵列。

泄漏电流的各种因素会导致 DRAM 单元在 10~100 毫秒时间内失去电荷。幸运的是，计算机运行的时钟周期是以纳秒来衡量的，这个保持时间已经很长了。存储器系统必须周期性地通过读出然后写回来刷新存储器的每个位。有些系统也使用错误纠正码，其中计算机的字会被多编码几个位（例如，32 位的字可能用 38 位来编码），这样一来，电路可以发现并纠正一个字中任何单个的错误位。

图 6.2 总结了 SRAM 和 DRAM 存储器的特性。只要有电，SRAM 就是持续的。与 DRAM 不同，它不需要刷新。SRAM 的存取比 DRAM 快。SRAM 对诸如光和电噪音这样的干扰不敏感。代价是 SRAM 单元比 DRAM 单元使用更多的晶体管，因而没那么密集，而且更贵，功耗更大。

	每位晶体管数	相对访问时间	持续的	敏感的?	相对花费	应用
SRAM	6	1X	Yes	No	100X	高速缓存存储器
DRAM	1	10X	No	Yes	1X	主存，帧缓冲区

图 6.2 DRAM 和 SRAM 存储器的特性

常规的 DRAM

DRAM 芯片中的单元（位）被分成 d 个*超单元*（supercell），每个超单元都是由 w 个 DRAM 单元组成的。一个 $d \times w$ 的 DRAM 总共存储了 dw 位信息。超单元被组织成一个 r 行 c 列的长方形阵列，这里 $rc=d$。每个超单元有形如(i,j)的地址，这里 i 表示行，而 j 表示列。

例如，图 6.3 展示的是一个 16×8 的 DRAM 芯片的组织，有 $d=16$ 个超单元，每个超单元有 $w=8$ 位，$r=4$ 行，$c=4$ 列。带阴影的方框表示地址(2,1)处的超单元。信息通过称为管脚（pin）的外部连接器流入和流出芯片。每个管脚携带一个一位的信号。图 6.3 给出了两组管脚：8 个 data 管脚，它们能传送一个字节到芯片或从芯片传出一个字节，以及 2 个 addr 管脚，它们携带 2 位的行和列超单元地址。其他携带控制信息的管脚没有显示出来。

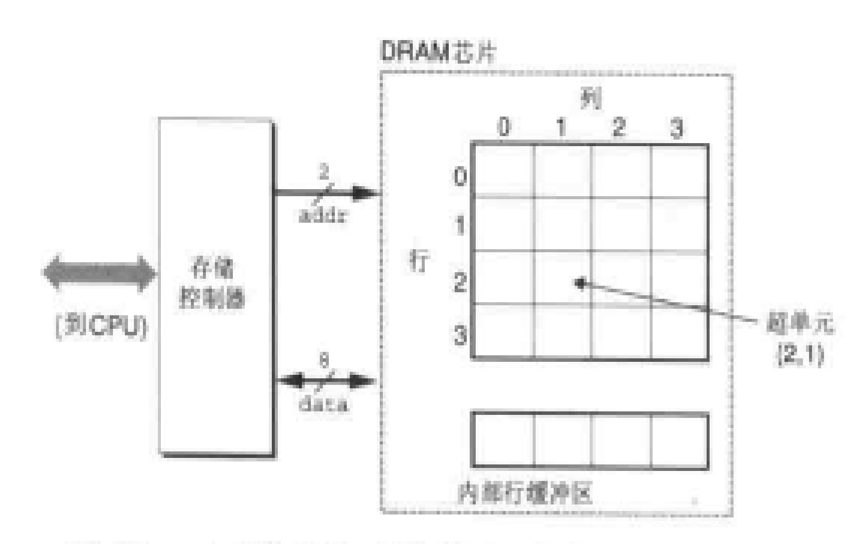


图 6.3 一个 128 位 16×8 的 DRAM 芯片的高级视图

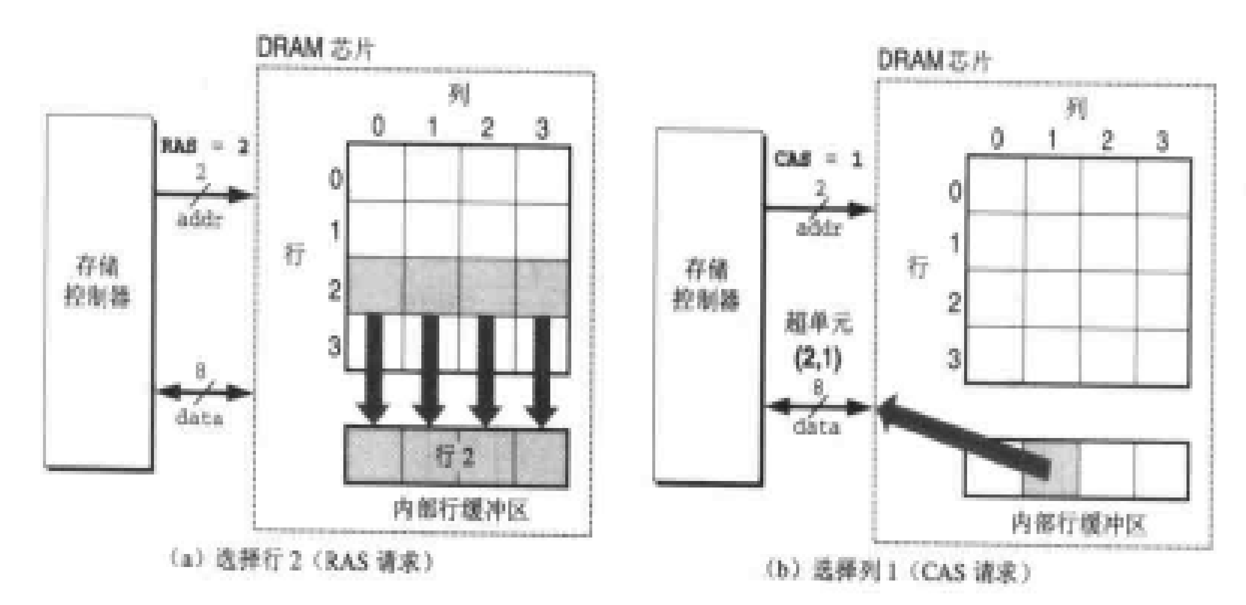
旁注：关于术语的注释

存储领域从来没有为 DRAM 的阵列元素确定一个标准的名字。计算机构架师倾向于称之为"单元（cell）"，使这个术语具有 DRAM 存储单元之意。电路设计者倾向于称之为"字（word）"，使之具有主存一个字之意。为了避免混淆，我们采用了无歧义的术语"超单元（supercell）"。

每个 DRAM 芯片被连接到某个称为*存储控制器*的电路，这个电路可以一次传送 w 位到每个 DRAM 芯片或一次从每个 DRAM 芯片传出 w 位。为了读出超单元(i,j)的内容，存储控制器将行地址 i 发送到 DRAM，然后是列地址 j。DRAM 把超单元(i,j)的内容发回给控制器作为响应。行地址 i 被称为 RAS（Row Access Strobe，行访问选通脉冲）请求，列地址 j 被称为 CAS（Column Access Strobe，列访问选通脉冲）请求。注意 RAS 和 CAS 请求共享同样的 DRAM 地址管脚。

例如，要从图 6.3 中的 16×8 的 DRAM 中读出超单元(2,1)，存储控制器发送行地址 2，如图 6.4（a）所示。DRAM 的响应是将行 2 的整个内容都拷贝到一个内部的行缓冲区。接下来，存储控制

器发送列地址 1，如图 6.4（b）所示。DRAM 的响应是从行缓冲区拷贝出超单元(2,1)中的 8 位，并把它们发送到存储控制器。



（a）选择行 2（RAS 请求） （b）选择列 1（CAS 请求）

图 6.4 读一个 DRAM 超单元的内容

电路设计者将 DRAM 组织成二维阵列而不是线性数组的一个原因是降低芯片上地址管脚的数量。例如，如果我们示例的 128 位 DRAM 被组织成一个 16 个超单元的线性数组，地址为 0～15，那么芯片会需要 4 个地址管脚而不是 2 个。二维阵列组织的缺点是必须分两步发送地址，这增加了访问时间。

存储器模块

DRAM 芯片包装在存储器模块（memory module）中，它插到主板的扩展槽上。常见的包装包括 168 个管脚的*双列直插存储器模块*（Dual Inline Memory Module，DIMM），它以 64 位为块传送数据到存储控制器和从存储控制器传出数据，还包括 72 个管脚的*单列直插存储器模块*（Single Inline Memory Module，SIMM），它以 32 位为块传送数据。

图 6.5 展示了一个存储器模块的基本思想。示例模块用八个 64Mbit 的 8M×8 的 DRAM 芯片，总共存储 64MB（兆字节），这八个芯片编号为 0～7。每个超单元存储主存的一个字节，而用相应超单元地址为(i,j)的八个超单元来表示主存中字节地址 A 处的 64 位双字[1]。在图 6.5 中的示例中，DRAM 0 存储第一个（低位）字节，DRAM 1 存储下一个字节，依此类推。

要取出存储器地址 A 处的一个 64 位双字，存储控制器将 A 转换成一个超单元地址(i,j)，并将它发送到存储器模块，然后存储器模块再将 i 和 j 广播到每个 DRAM。作为响应，每个 DRAM 输出它的(i,j)超单元的 8 位内容。模块中的电路收集这些输出，并把它们合并成一个 64 位双字，再返回给存储控制器。

通过将多个存储器模块连接到存储控制器，能够聚合主存。在这种情况中，当控制器收到一个地址 A 时，控制器选择包含 A 的模块 k，将 A 转换成它的(i,j)的形式，并将(i,j)发送到模块 k。

1 IA32 会称 64 位为"四字"。

练习题 6.1

接下来，r 表示一个 DRAM 阵列中的行数，c 表示列数，b_r 表示行寻址所需的位数，b_c 表示列寻址所需的位数。对于下面每个 DRAM，确定 2 的幂数的数组维数，使得 $\max(b_r, b_c)$ 最小，$\max(b_r, b_c)$ 是对数组的行或列寻址所需的位数中较大的值。

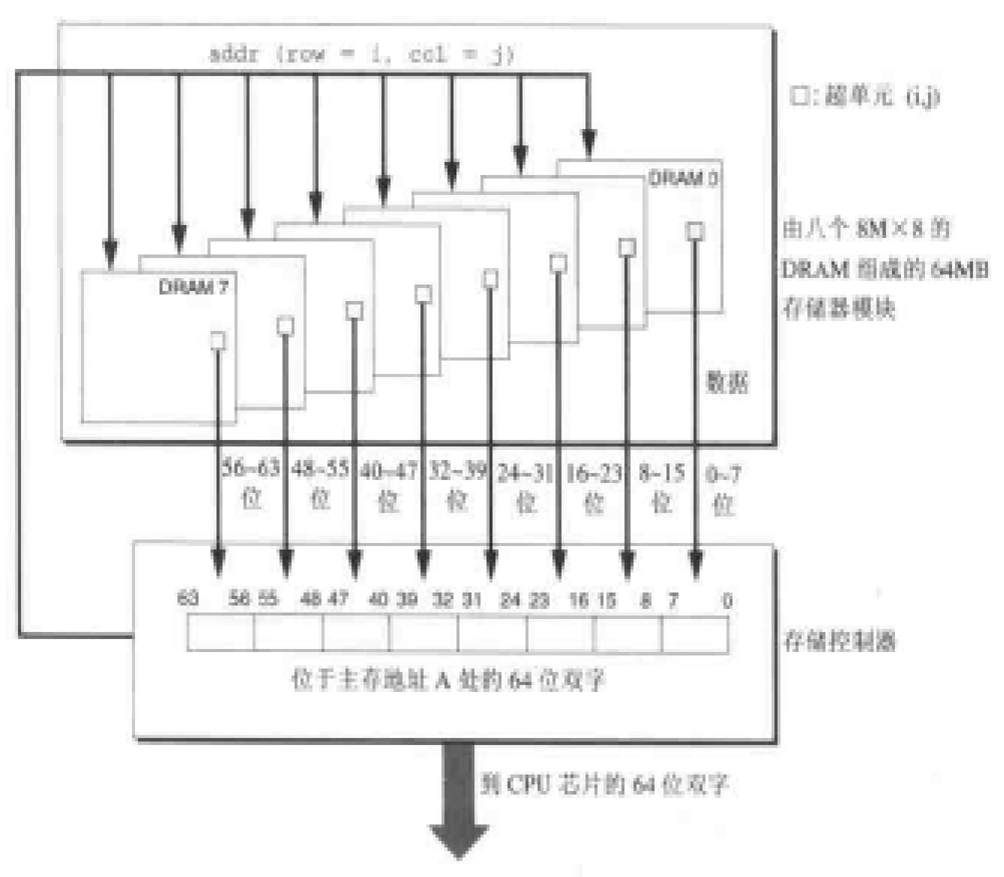


图 6.5 读一个存储器模块的内容

组织	r	c	b_r	b_c	$\max(b_r, b_c)$
16 × 1					
16 × 4					
128 × 8					
512 × 4					
1024 × 4					

增强的 DRAM

有许多种 DRAM 存储器，而生产厂商试图跟上迅速增长的处理器速度，市场上会定期推出新的种类。每种都是基于传统的 DRAM 单元，并进行了一些优化，改进了访问基本 DRAM 单元的速度。

- FPM DRAM（fast page mode DRAM，快页模式 DRAM）。传统的 DRAM 将超单元的一整行拷贝到它的内部行缓冲区中，使用一个，然后丢弃剩余的。FPM DRAM 允许对同一行连续的访问可以直接从行缓冲区得到服务，从而改进了这一点。例如，要从一个传统的 DRAM 的行 i 中读四个超单元，存储控制器必须发送四个 RAS/CAS 请求，即使是行地址 i 在每个情况中

都是一样的。要从一个FPM DRAM的同一行中读取超单元，存储控制器发送第一个RAS/CAS请求，后面跟三个CAS请求。开始的RAS/CAS请求将行i拷贝到行缓冲区，并返回第一个超单元。接下来三个超单元直接从行缓冲区获得服务，因此比开始的超单元更快。

- EDO DRAM（extended data out DRAM，*扩展数据输出DRAM*）。FPM DRAM的一个增强的形式，它允许单独的CAS信号在时间上靠得更紧密一点。
- SDRAM（synchronous DRAM，*同步DRAM*）。就它们与存储控制器通信使用一组显式的控制信号来说，常规的、FPM和EDO DRAM都是异步的。SDRAM用与驱动存储控制器相同的外部时钟信号的上升沿来代替许多这样的控制信号。我们不会深入讨论细节，最终效果就是SDRAM能够比那些异步的存储器更快地输出它的超单元的内容。
- DDR SDRAM（double data-rate synchronous DRAM，*双倍数据速率同步DRAM*）。DDR SDRAM是对SDRAM的一种增强，它通过使用时钟的两个边沿作为控制信号，从而使DRAM的速度翻倍。
- Rambus DRAM（RDRAM）。这是另一种私有技术，它的最大带宽比DDR SDRAM的更高。
- VRAM RAM（Video RAM，*视频*）。它用在图形系统的帧缓冲区中。VRAM的思想与FPM DRAM类似。两个主要区别是：①VRAM的输出是通过依次对内部缓冲区的整个内容进行移位得到的；②VRAM允许对存储器并行地读和写。因此，系统可以在写下一次更新的新值（写）的同时，用帧缓冲区中的像素刷屏幕（读）。

旁注：DRAM技术流行的历史

直到1995年，大多数PC都是用FPM DRAM制造的。1996~1999年，EDO DRAM在市场上占据了优势，而FPM DRAM几乎销声匿迹了。SDRAM最早出现在1995年在高端系统中，而到2002年，大多数PC都是用SDRAM和DDR SDRAM制造的。

非易失性储器

如果断电，DRAM和SRAM会丢失它们的信息，从这个意义上说，它们是*易失的*（volatile）。另一方面，*非易失性存储器*（nonvolatile memory）即使是在关电后，仍然保存着它们的信息。有很多种非易失性存储器。由于历史原因，虽然ROM中有的类型既可以读也可以写，但是它们整体上都被称为ROM（read-only memory，*只读存储器*）。ROM是以它们能够被重编程（写）的次数和对它们进行重编程所用的机制来区分的。

PROM（programmable ROM，*可编程ROM*）只能被编程一次。PROM包括一种熔丝（fuse），每个存储器单元只能用高电流熔断一次。

EPROM（erasable programmable ROM，*可擦写可编程ROM*）有一个透明的石英窗口，允许光到达存储单元。紫外线光照射过窗口，EPROM单元就被清除为0。对EPROM编程是通过使用一种把1写入EPROM的特殊设备来完成的。EPROM能够被擦除和重编程的次数的数量级可以达到1000次。EEPROM（electrically erasable PROM，*电子可擦除PROM*）类似于EPROM，但是它不需要一个物理上独立的编程设备，因此可以直接在印制电路卡上编程。EEPROM能够被编程的次数的数量级可以达到10^5次。*闪存*（flash memory）是一类小的非易失性存储器，基于EEPROM，它可以插入到桌面机器、手持设备或视频游戏控制台，以及从上面拔下来。

存储在ROM设备中的程序通常被称为*固件*（firmware）。当一个计算机系统通电以后，它会运

行存储在 ROM 中的固件。一些系统在固件中提供了少量基本的输入和输出函数——例如，PC 的 BIOS（基本输入/输出系统）例程。复杂的设备，像图形卡和磁盘驱动器，也依赖固件来翻译来自 CPU 的 I/O（输入/输出）请求。

访问主存

数据流通过称为总线（bus）的共享电路在处理器和 DRAM 主存之间来来回回。每次 CPU 和主存之间的数据传送都是通过一系列步骤来完成的，这些步骤称为**总线事务**（bus transaction）。**读事务**（read transaction）从主存传送数据到 CPU。**写事务**（write transaction）从 CPU 传送数据到主存。

总线是一组并行的导线，能携带地址、数据和控制信号。取决于总线设计，数据和地址信号可以共享同一组导线，也可以使用不同的。同时，两个以上的设备也能共享同一个总线。控制线携带的信号会同步事务，并标识出当前正在被执行的事务的类型。例如，当前关注的这个事务是到主存的吗？还是到诸如磁盘控制器这样的其他 I/O 设备？这个事务是读还是写？总线上的信息是地址还是数据项？

图 6.6 展示了一台典型的桌面系统的结构。主要部件是 CPU 芯片、我们将称为 I/O 桥接器（I/O bridge）的芯片组（其中包括存储控制器），以及组成主存的 DRAM 存储器模块。这些部件由一对总线连接起来的，其中一条总线是**系统总线**（system bus），它将 CPU 连接到 I/O 桥接器，另一条总线是**存储器总线**（memory bus），它将 I/O 桥接器连接到主存。

I/O 桥接器将系统总线的电信号翻译成存储器总线的电信号。正如我们看到的那样，I/O 桥接器也将系统总线和存储器总线连接到 I/O 总线，像磁盘和图形卡这样的 I/O 设备共享 I/O 总线。不过现在，我们将注意力集中在存储器总线上。

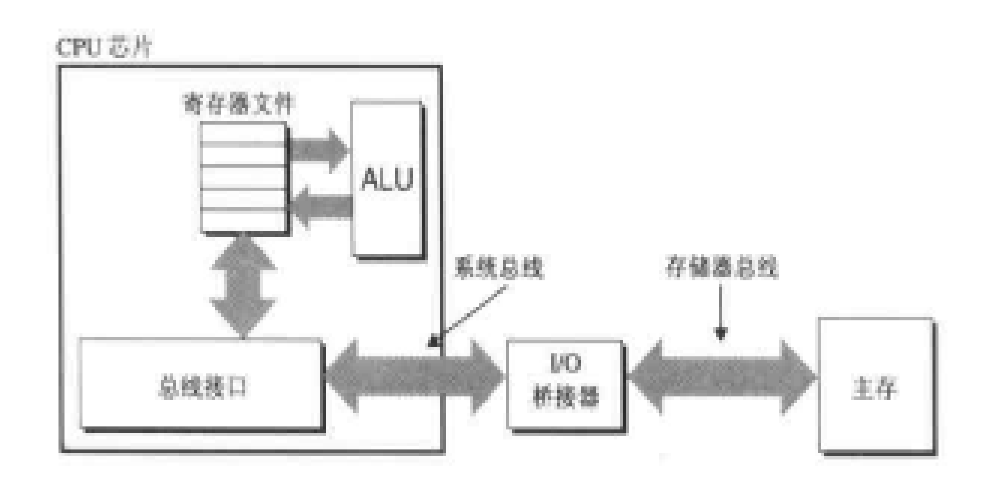


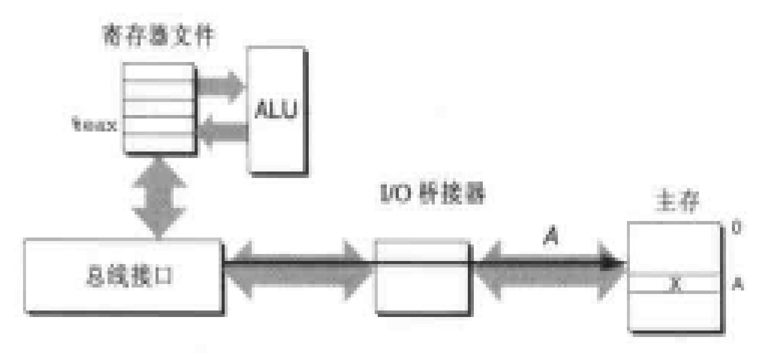
图 6.6 典型的连接 CPU 和主存的总线结构

考虑当 CPU 执行一个加载操作时会发生什么

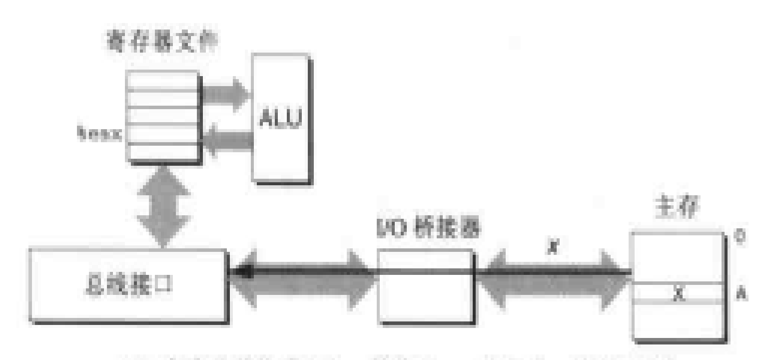
```
movl A, %eax
```

这里，地址 A 的内容被加载到寄存器%eax 中。CPU 芯片上称为**总线接口**（bus interface）的电路发起总线上的读事务。读事务是由三个步骤组成的。首先，CPU 将地址 A 放到系统总线上。I/O 桥接器将信号传递到存储器总线，如图 6.7（a）。接下来，主存感觉到存储器总线上的地址信号，从

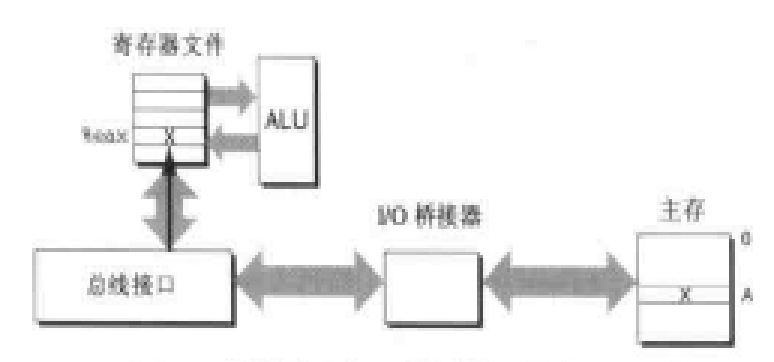
存储器总线读地址，从 DRAM 取出数据字，并将数据写到存储器总线。I/O 桥接器将存储器总线信号翻译成系统总线信号，然后传递到系统总线，如图 6.7（b）。最后，CPU 感觉到系统总线上的数据，从总线上读数据，并将数据拷贝到寄存器%eax，如图 6.7（c）。



（a）CPU 将地址 A 放到存储器总线



（b）主存从总线读出 A，接收字 x，然后将 x 放到总线上



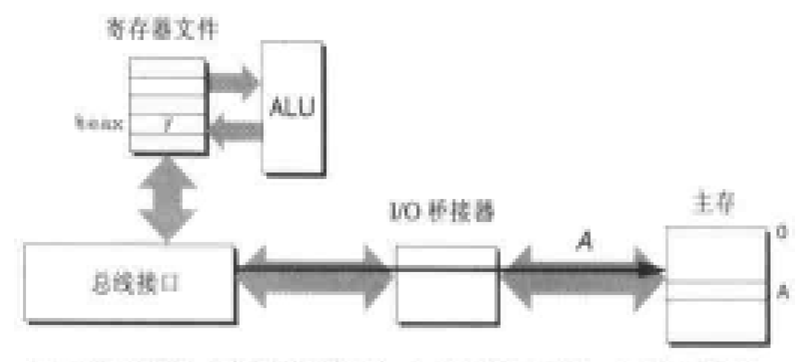
（c）CPU 从总线读出字 x，并将它拷贝到寄存器%eax 中

图 6.7 加载操作 movl A, %eax 的存储器读事务

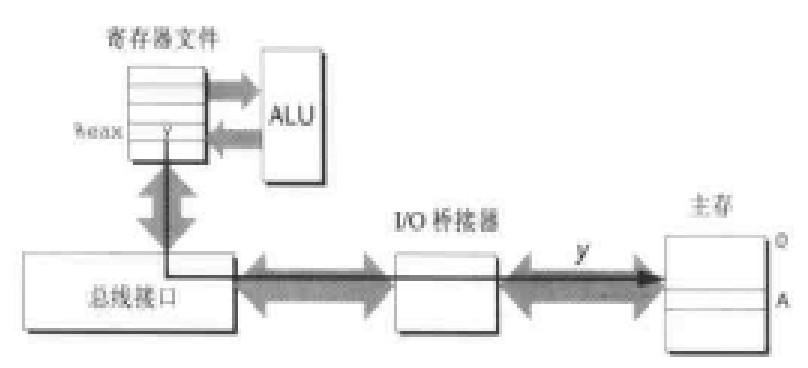
相反地，当 CPU 执行一个存储操作时

```
movl %eax, A
```

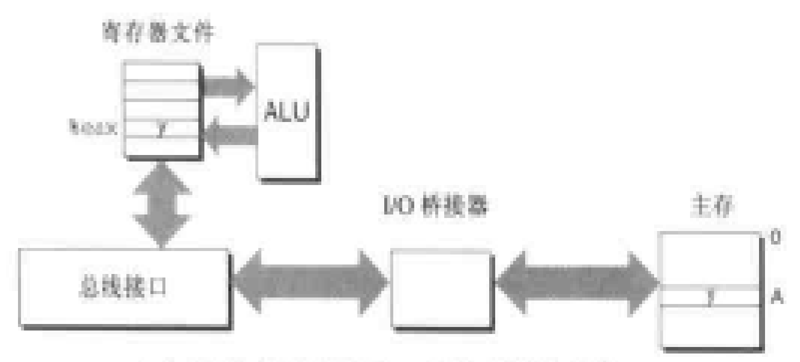
这里，寄存器%eax 的内容被写到地址 A，CPU 发起写事务。同样，有三个基本步骤。首先，CPU 将地址放到系统总线上。存储器从主存总线读出地址，并等待数据到达，如图 6.8（a）。接下来，CPU 将%eax 中的数据字拷贝到系统总线，如图 6.8（b）。最后，主存从存储器总线读出数据字，然后再将这些位存储到 DRAM 中，如图 6.8（c）。



(a) CPU 将地址 A 放到存储器总线，主存读出这个地址，并等待数据字



(b) CPU 将数据字 y 放到总线上



(c) 主存从总线读数据字 y，并将它存储在地址 A

图 6.8 存储操作 movl %eax, A 的存储器写事务

6.1.2 磁盘存储

磁盘是广为应用的保存大量数据的存储设备，存储数据的数量级可以达到几十到几百千兆字节，而基于 RAM 的存储器只能有几百或几千兆字节。不过，从磁盘上读信息需要几毫秒，比从 DRAM 读慢了 10 万倍，比从 SRAM 读慢了 100 万倍。

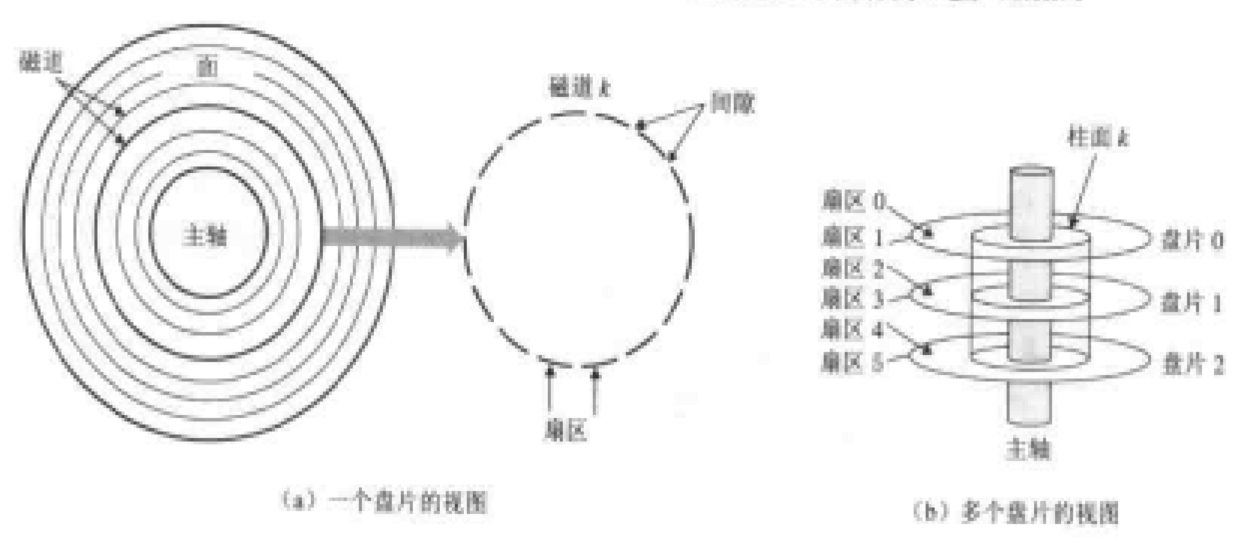
磁盘构造

磁盘是由**盘片**（platter）构成的。每个盘片有两面，**表面**（surface）覆盖着磁性记录材料。盘片中间有一个可以旋转的**主轴**（spindle），它使得盘片以固定的**旋转速率**（rotational rate）旋转，通常是 5 400～15 000RPM（revolution per minute，**转每分钟**）。磁盘通常包含一个或多个这样的盘片，且装在一个密封的容器内。

图 6.9（a）展示了一个典型的磁盘表面的结构。每个表面是由一组称为**磁道**（track）的同心圆

组成的，且每个磁道被划分为一组扇区（sector）。每个扇区包含相等数量的数据位（通常是 512 字节），这些数据编码在扇区上的磁性材料中。扇区之间由一些间隙（gap）分隔开，这些间隙中不存储数据位。间隙存储用来标识扇区的格式化位。

磁盘是由一个或多个叠放在一起的盘片组成的，它们放在一个密封的包装里，如图 6.9（b）所示。整个装置通常被称为**磁盘驱动器**（disk drive），虽然我们通常简称为**磁盘**（disk）。



（a）一个盘片的视图　　（b）多个盘片的视图

图 6.9　磁盘构造

磁盘制造商通常用术语**柱面**（cylinder）来描述多个盘片驱动器的构造，这里，柱面是所有盘片表面上到中心主轴的距离相等的磁道的集合。例如，如果一个驱动器有三个盘片、六个面，每个面上的磁道的编号都是一致的，那么柱面 k 就是六个磁道 k 的集合。

磁盘容量

一个磁盘上可以记录的最大位数被称为它的**最大容量**，或者简称为**容量**。磁盘容量是由以下技术因素决定的：

- 记录密度（recording density）（位/英寸）：磁道一英寸的段中可以放入的位数。
- 磁道密度（track density）（道/英寸）：从盘片中心出发半径为一英寸的段内可以有的磁道数。
- 面密度（areal density）（位/平方英寸）：记录密度与磁道密度的乘积。

磁盘制造商不懈地努力以增加面密度（从而增加容量），而面密度每隔几年就会翻倍。最初的磁盘，是在面密度很低的时代设计的，将每个磁道分为数目相同的扇区，扇区的数目是由最内的磁道能记录的扇区数决定的。为了保持每个磁道有固定的扇区数，越往外的磁道扇区隔得越开。在面密度相对比较低的时候，这种方法很合理。不过，随着面密度的提高，扇区之间的间隙（那里没有存储数据位）变得不可接收的大了。因此，现代大容量磁盘使用一种称为**多区记录**（multiple zone recording）的技术。在这种技术中，磁道集合被分成了不相交的子集合，称为**记录区**（recording zone）。每个区包含一组连续的磁道。一个区中的每个磁道都有相同数量的扇区，这个扇区的数量是由该区中最里面的磁道所包含的扇区数确定的。注意，软盘仍然使用的是老式的方法，每个磁道的扇区数是常数。

下面的公式给出了一个磁盘的容量：

$$\text{Disk capacity} = \frac{\#bytes}{sector} \times \frac{average\ \#\ sectors}{track} \times \frac{\#\ tracks}{surface} \times \frac{\#\ surfaces}{platter} \times \frac{\#\ platters}{disk}$$

例如，假设我们有一个磁盘，有 5 个盘片，每个扇区 512 字节，每个面 20 000 条磁道，每条磁道平均 300 个扇区。那么这个磁盘的容量是：

$$\begin{aligned}\text{Disk capacity} &= \frac{512\ bytes}{sector} \times \frac{300\ sectors}{track} \times \frac{20\,000\ tracks}{surface} \times \frac{2\ surfaces}{platter} \times \frac{5\ platters}{disk} \\ &= 30\,720\,000\,000\ bytes \\ &= 30.72\ \text{GB}\end{aligned}$$

注意，制造商是以千兆字节（GB）为单位来表达磁盘容量的，这里 $1\text{GB} = 10^9$ 字节。

旁注：一吉字节有多大？

不幸地，像 K（kilo）、M（mega）和 G（giga）这样的前缀的含义依赖于上下文。对于与 DRAM 和 SRAM 容量相关的单位，通常 $K = 2^{10}$，$M = 2^{20}$，而 $G = 2^{30}$。对于与像磁盘和网络这样的 I/O 设备容量相关的单位，通常 $K = 10^3$，$M = 10^6$，而 $G = 10^9$。速率和吞吐量常常也使用这些前缀。

幸运地，对于我们通常依赖的封底（back-of-the-envelope）估计值，无论是哪种假设在实际中都工作得很好。例如，$2^{20} = 1\,048\,576$ 和 $10^6 = 1\,000\,000$ 之间相对差别很小：$(2^{20} - 10^6) / 10^6 \approx 5\%$。类似地，对于 $2^{30} = 1\,073\,741\,824$ 和 $10^9 = 1\,000\,000\,000$：$(2^{30} - 10^9) / 10^9 \approx 7\%$。

练习题 6.2

计算这样一个磁盘的容量，它有 2 个盘片，10 000 个柱面，每条磁道平均有 400 个扇区，而每个扇区有 512 字节。

磁盘操作

磁盘用连接到一个传动臂（actuator arm）的读/写头（read/write head）来读写存储在磁性表面的位，如图 6.10（a）所示。通过沿着半径轴移动这个传动臂，驱动器可以将读/写头定位在盘面上的任何磁道上。这样的机械运动称为寻道（seek）。一旦读/写头定位到了期望的磁道上，那么当磁道上的每个位通过它的下面时，读/写头可以感知到这个位的值（读该位），也可以修改这个位的值（写该位）。有多个盘片的磁盘针对每个盘面都有一个独立的读/写头，如图 6.10（b）所示。读/写头垂直排列，一致行动。在任何时刻，所有的读/写头都位于同一个柱面上。

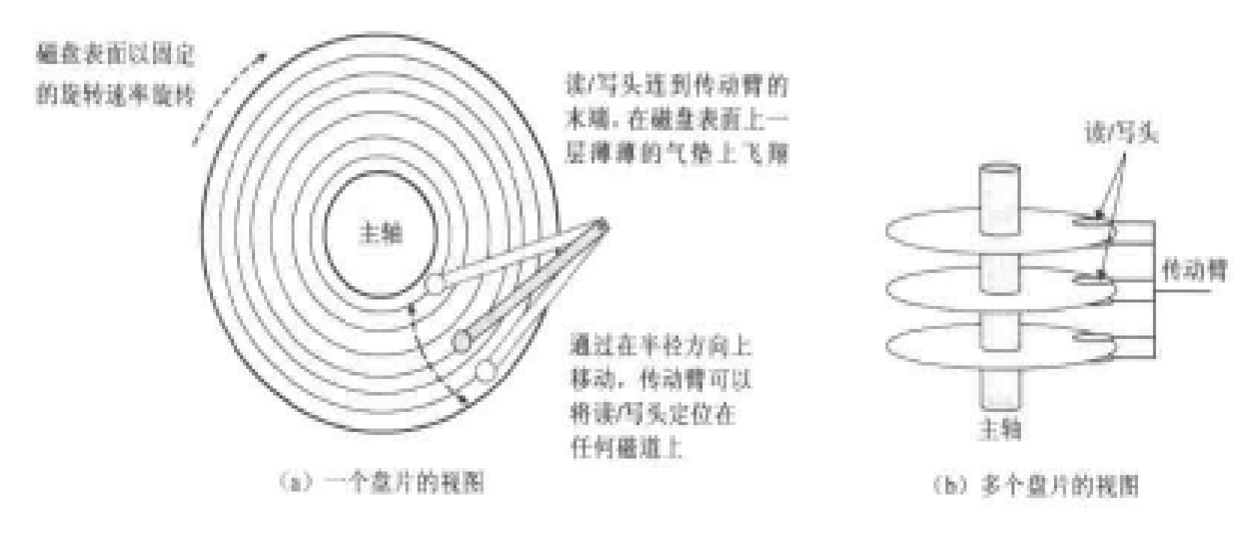


图 6.10 磁盘的动态特性

在传动臂末端的读/写头在磁盘表面高度大约 0.1 微米处的一层薄薄的气垫上飞翔，速度大约为 80km/h。在这样小的间隙里，盘面上一粒微小的灰尘都像一块巨石。如果读/写头碰到了这样的一块巨石，读/写头会停下来，撞到盘面——所谓的读/写头冲撞（head crash）。为此，磁盘总是密封包装的。

磁盘以扇区大小的块来读写数据。对扇区的访问时间（access time）有三个主要的部分：寻道时间（seek time）、旋转时间（rotational latency）和传送时间（transfer time）：

- **寻道时间**：为了读取某个目标扇区的内容，传动臂首先将读/写头定位到包含目标扇区的磁道上。移动传动臂所需的时间称为寻道时间。寻道时间 T_{seek} 依赖于传动臂以前的位置和传动臂在盘面上移动的速度。现代驱动器中平均寻道时间 $T_{avg\ seek}$ 是通过对几千次对随机扇区的寻道求平均值来测量的，通常为 6～9ms。一次寻道的最大时间 $T_{max\,seek}$ 可以高达 20ms。
- **旋转时间**：一旦读/写头定位到了期望的磁道，驱动器等待目标扇区的第一个位旋转到读/写头下。这个步骤的性能依赖于当读/写头到达目标扇区时盘面的位置，和磁盘的旋转速度。在最坏的情况下，读/写头刚刚错过了目标扇区，必须等待磁盘转一整圈。因此，最大旋转时间，以秒为单位，是

$$T_{max\ rotation} = \frac{1}{RPM} \times \frac{60\ secs}{1\ min}$$

平均旋转时间 $T_{avg\ rotation}$ 是 $T_{max\ rotation}$ 的一半。

- **传送时间**：当目标扇区的第一个位位于读/写头下时，驱动器就可以开始读或者写该扇区的内容了。一个扇区的传送时间依赖于旋转速度和每条磁道的扇区数目。因此，我们可以粗略地估计一个扇区以秒为单位的平均传送时间如下

$$T_{avg\ transfer} = \frac{1}{RPM} \times \frac{1}{(average\#\ sectors/track)} \times \frac{60\ secs}{1\ min}$$

我们可以估计访问一个磁盘扇区内容的平均时间为平均寻道时间、平均旋转时间和平均传送时间的和。例如，考虑一个有如下参数的磁盘：

参 数	值
旋转速率	7 200RPM
$T_{avg\ seek}$	9ms
每条磁道的平均扇区数	400

对于这个磁盘，平均旋转时间（以 ms 为单位）是

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ secs/7200\ RPM) \times 1000\ ms/sec \\ &\approx 4\ ms \end{aligned}$$

平均传送时间是

$$\begin{aligned} T_{avg\ transfer} &= 60/7200\ RPM \times 1/400\ sectors/track \times 1000\ ms/sec \\ &\approx 0.02\ ms \end{aligned}$$

总之，整个估计的访问时间是

$$
\begin{aligned}
T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\
&= 9\ ms + 4\ ms + 0.02\ ms \\
&= 13.02\ ms
\end{aligned}
$$

这个例子说明了一些很重要的问题：

- 访问一个磁盘扇区中 512 字节的时间主要是寻道时间和旋转时间。访问扇区中的第一个字节用了很长时间，但是剩下的字节几乎不用时间。
- 因为寻道时间和旋转时间大致相等的，所以将寻道时间乘 2 是估计磁盘访问时间的简单而合理的方法。
- 对存储在 SRAM 中的双字的访问时间大约是 4ns，对 DRAM 的访问时间是 60ns。因此，从存储器中读一个 512 字节扇区大小的块的时间对 SRAM 来说大约是 256ns，对 DRAM 来说大约是 4000ns。磁盘访问时间（大约 10ms）比 SRAM 大约大 40000 倍，比 DRAM 大约大 2500 倍。如果我们比较访问一个单字的时间，这些访问时间的差别会更大。

练习题 6.3

估计访问下面这个磁盘上一个扇区的访问时间（以 ms 为单位）。

参 数	值
旋转速率	15 000RPM
$T_{avg\ seek}$	8ms
每条磁道的平均扇区数	500

逻辑磁盘块

正如我们看到的那样，现代磁盘构造复杂，有多个盘面，这些盘面上有不同的记录区。为了对操作系统隐藏这样的复杂性，现代磁盘将它们的构造简化为一个 b 个扇区大小的**逻辑块**的序列，编号为 0,1,…,b-1。磁盘中有一个小的硬件/固件设备，称为**磁盘控制器**，维护着逻辑块号和实际（物理）磁盘扇区之间的映射关系。

当操作系统想要执行一个 I/O 操作时，例如读一个磁盘扇区的数据到主存，操作系统会发送一个命令到磁盘控制器，让它读某个逻辑块号。控制器上的固件执行一个快速表查找，将一个逻辑块号翻译成一个（*盘面，磁道，扇区*）的三元组，这个三元组惟一地标识了对应的物理扇区。控制器上的硬件解释这个三元组，将 I/O 头移动到适当的柱面，等待扇区移动到 I/O 头下，将 I/O 头感知到的位放到控制器上的一个小缓冲区中，然后将它们拷贝到主存中。

旁注：格式化的磁盘容量

在磁盘可以存储数据之前，它必须被磁盘控制器格式化。这包括用标识扇区的信息填写扇区之间的间隙，标识出表面有故障的柱面并且不使用它们，以及在每个区中预留出一组柱面作为备用。如果区中一个柱面在磁盘使用过程中坏掉了，就可以使用这些备用的柱面。因为存在着这些备用的柱面，所以磁盘制造商所说的格式化容量比最大容量要小。

访问磁盘

像图形卡、监视器、鼠标、键盘和磁盘这样的设备都是通过诸如 Intel 的 PCI（Peripheral Component

Interconnect，外围设备互连）总线这样的 I/O 总线连接到 CPU 和主存的。同系统总线和存储器总线不同（它们是与 CPU 相关的），诸如 PCI 这样的 I/O 总线设计成与底层 CPU 无关。例如，PC 和 Macintosh 都可以使用 PCI 总线。图 6.11 展示了一个典型的 I/O 总线结构（以 PCI 为模型），它连接了 CPU、主存和 I/O 设备。

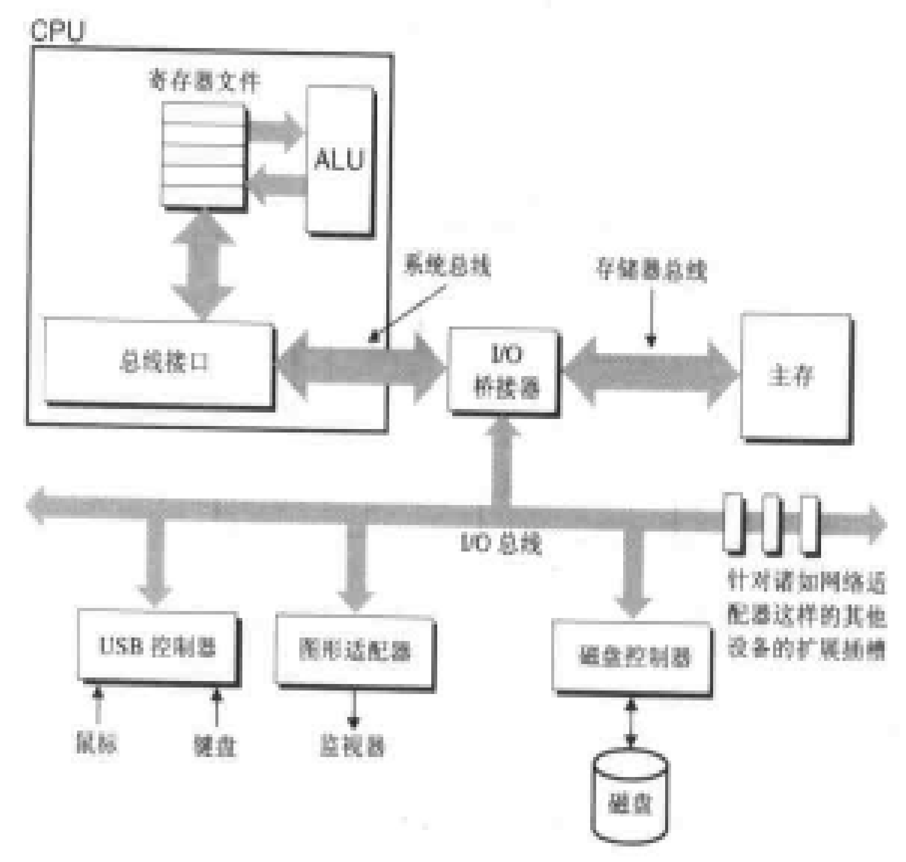


图 6.11 典型的总线结构，它连接 CPU、主存和 I/O 设备

虽然 I/O 总线比系统总线和存储器总线慢，但是它可以容纳种类繁多的第三方 I/O 设备。例如，在图 6.11 中，有三个不同类型的设备连接到总线：

- USB（Universal Serial Bus，通用串行总线）控制器是一个将设备连接到 USB 的电路。USB 的吞吐率可以达到 12Mbit/s，是为慢速或中速串行设备设计的，例如键盘、鼠标、调制解调器、数码相机、操纵杆、CD-ROM 驱动器和打印机。
- 图形卡（或适配器）包含硬件和软件逻辑，它们负责代表 CPU 在显示器上画像素。
- 磁盘控制器包含硬件和软件逻辑，它们用来代表 CPU 读写磁盘数据。

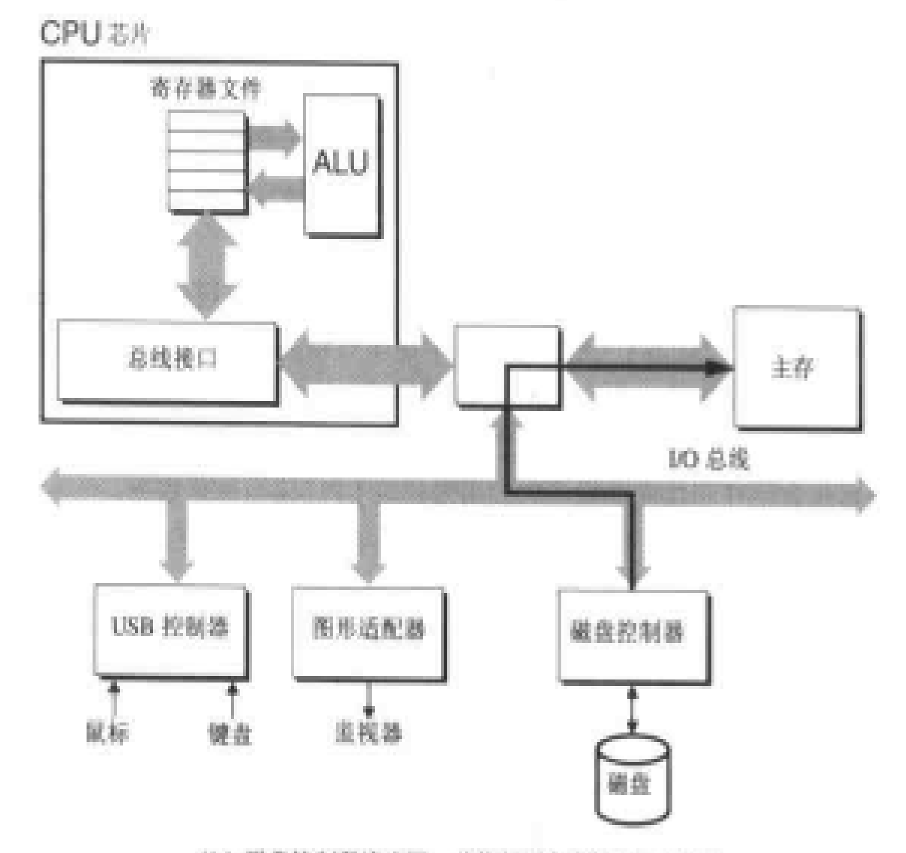
其他的设备，例如网络适配器，可以通过将适配器插入到主板上空的扩展槽中，从而连接到 I/O 总线，这些插槽提供了到总线的直接电路连接。

虽然详细描述 I/O 设备是如何工作的以及如何对它们进行编程，超出了我们讨论的范围，但是我们可以给你一个概要的描述。例如，图 6.12 总结了当 CPU 从磁盘读数据时发生的步骤。

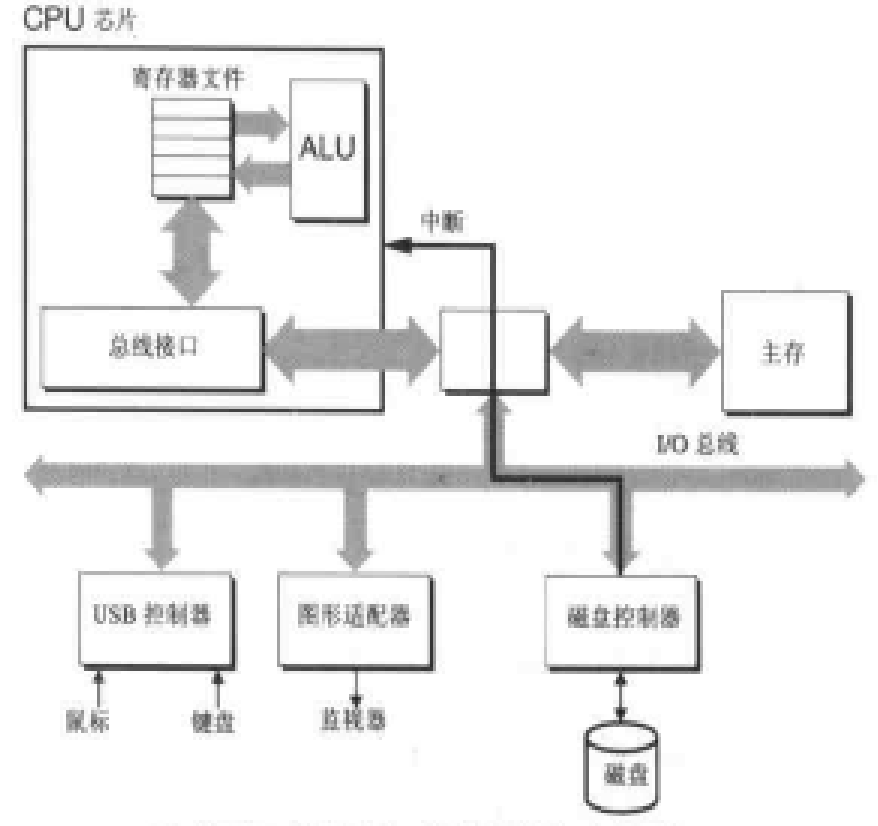
CPU 使用一种称为存储器映射 I/O（memory-mapped I/O）的技术来向 I/O 设备发射命令，如图 6.12（a）所示。在使用存储器映射 I/O 的系统中，地址空间中有一块地址是为与 I/O 设备通信保留的。每个这样的地址称为一个 I/O 端口（I/O port）。当一个设备连接到总线时，它与一个或多个端

口相关联（或它被映射到一个或多个端口）。

（a）CPU 通过将命令、逻辑块号和目的存储器地址写到与磁盘相关联的存储器映射地址，发起一个磁盘读



（b）磁盘控制器读扇区，并执行到主存的 DMA 传送



（c）当 DMA 传送完成时，磁盘控制器以一个中断通知 CPU

图 6.12 读一个磁盘扇区

作为一个简单的例子，假设磁盘控制器被映射到端口 0xa0。随后，CPU 可能通过执行三个对地址 0xa 的存储指令，发起磁盘读：第一条指令是发送一个命令字，它告诉磁盘发起一个读，还发送了其他的参数，例如当读完成时，是否中断 CPU（我们会在 8.1 节中讨论中断）。第二条指令指明应该读的逻辑块号。第三条指令指明应该存储磁盘扇区内容的主存地址。

当 CPU 发起了请求之后，在磁盘执行读的时候，它通常会做些其他的工作。回想一下，一个 1GHz 的处理器时钟周期为 1ns，在用来读磁盘的 16ms 时间里，它潜在地可能执行 1600 万条指令。在传输进行时，只是简单地等待，什么都不做，是一种极大的浪费。

在磁盘控制器收到来自 CPU 的读命令之后，它将逻辑块号翻译成一个扇区地址，读该扇区的内容，然后将这些内容直接传送到主存，不需要 CPU 的干涉，如 6.12（b）所示。这个设备可以自己执行读或者写总线事务，而不需要 CPU 干涉的过程，称为 DMA（direct memory access，*直接存储器访问*）。这种数据传送称为 DMA *传送*（DMA transfer）。

在 DMA 传送完成，磁盘扇区的内容被安全地存储在主存中以后，磁盘控制器通过给 CPU 发送一个中断信号来通知 CPU，如图 6.12（c）所示。基本思想是中断会发信号到 CPU 芯片的一个外部管脚上。这会导致 CPU 暂停它当前正在做的工作，跳转到一个操作系统函数。这个函数会记录下 I/O 已经完成，然后将控制返回到 CPU 被中断的地方。

旁注：一个商用磁盘的剖析

磁盘制造商在他们的网页上公布了许多高级技术信息。例如，如果我们访问 IBM Ultrastar 36LZX 磁

盘的网页，我们可以得到如图 6.13 所示的构造和性能信息。

构造属性	值
盘片	6
表面（头）	12
扇区大小	512 字节
区	11
柱面	15 110
记录密度（最大）	352 000 位/英寸
磁道密度	20 000 磁道/英寸
面密度	7 040Mb/平方英寸
格式化的容量	36GB

性能属性	值
旋转速率	10 000RPM
平均旋转时间	2.99ms
平均寻道时间	4.9ms
持续的传送速率	21～36MB/s

图 6.13 IBM Ultrastar 36LZX 的构造和性能

来源：www.storage.ibm.com。

磁盘制造商通常会忽略公布关于每个记录区构造的详细信息。不过，存储技术研究人员开发出了一个很有用的工具，称为 DIXtrac，它能自动发现大量关于 SCSI 磁盘构造和性能的低级信息[68]。例如，DIXtrac 能够发现我们示例的 IBM 磁盘详细的区构造，如图 6.14 所示。表中的每一行都描述了磁盘表面 11 个区中的某一个，关于该区中扇区数目、映射到该区中扇区的逻辑块的范围，以及该区中柱面的范围和数目。

区号	每磁道扇区数	起始逻辑块号	终止逻辑块号	起始柱面号	结束柱面号	每个区的柱面数
（外）0	504	0	2 292 096	1	380	380
1	476	2 292 097	11 949 751	381	2 078	1 698
2	462	11 949 752	19 416 566	2 079	3 430	1 352
3	420	19 416 567	36 409 689	3 431	6 815	3 385
4	406	36 409 690	39 844 151	6 816	7 523	708
5	392	39 844 152	46 287 903	7 524	8 898	1 375
6	378	46 287 904	52 201 829	8 899	10 207	1 309
7	364	52 201 830	56 691 915	10 208	11 239	1 032
8	352	56 691 916	60 087 818	11 240	12 046	807
9	336	60 087 819	67 001 919	12 047	13 768	1 722
（内）10	308	67 010 920	71 687 339	13 769	15 042	1 274

图 6.14 IBM Ultrastar 36LZX 的区图

来源：DIXtrac 自动磁盘驱动器描述工具[68]。

这个区图表述一步证实了一些关于 IBM 磁盘的有趣的事实。首先，靠外边的区（周长更长）比靠里面的区有更多的扇区。第二，每个区有比逻辑块更多的扇区（你可以自己检查一下）。未被使用的扇区形成一个备用柱面池。如果一个扇区上的记录材料坏了，磁盘控制器会自动地将该柱面上的逻辑块重映射到一个可用的备用柱面上。所以，我们看到，逻辑块的概念不仅能够提供给操作系统一个更简单的接口，还能够提供一层抽象，使得磁盘能够更健壮。就像我们在第 10 章中研究虚拟存储器时将会看到的那样，这种通用的抽象思想非常强大。

6.1.3 存储技术趋势

从我们对存储技术的讨论中，可以总结出几个很重要的思想：

- **不同的存储技术有不同的价格和性能折中。** SRAM 比 DRAM 快一点，而 DRAM 比磁盘要快很多。另一方面，快速存储总是比慢速存储要贵的。SRAM 每字节的造价比 DRAM 高，DRAM 的造价又比磁盘高得多。
- **不同存储技术的价格和性能属性以截然不同的速率变化着。** 图 6.15 总结了从 1980 年以来的存储技术的价格和性能属性，最早的 PC 是那一年提出的。这些数字是从以前的贸易杂志中挑选出来的。虽然它们是从非正式的调查中得到的，但是这些数字还是能揭示出一些有趣的趋势的。

  自从 1980 年以来，SRAM 技术的成本和性能基本上是以相同的速度改善的。访问时间下降了大约 100 倍，而每兆字节的成本下降了 200 倍，如图 6.15（a）所示。不过，DRAM 和磁盘的变化更大，而且不一致。DRAM 每兆字节的成本下降了 8000 倍所示几乎是四个数量级），而 DRAM 的访问时间只下降了大约 6 倍，如图 6.15（b）所示。磁盘技术有和 DRAM 相同的趋势，甚至于变化更大。从 1980 年以来，磁盘存储的每兆字节成本增长了 50 000 倍，访问时间改善得很少，只有 10 倍左右，如图 6.15（c）所示。这些惊人的长期趋势突出了存储器和磁盘技术的一个基本事实：增加密度（从而降低成本）比降低访问时间更容易。
- **DRAM 和磁盘访问时间滞后于 CPU 时钟周期时间。** 正如我们在图 6.15（d）中看到的那样，从 1980 年到 2000 年，CPU 时钟周期提高了 600 倍。相比于 CPU 性能，SRAM 的性能是稍差的，尽管 SRAM 的性能在保持增长。然而，DRAM 和磁盘性能与 CPU 性能之间的差距实际上是加大许多。图 6.16 清楚地表面了各种趋势，以半对数为标度（semi-log scale），画出了图 6.15 中的访问时间和时钟周期。

度量标准	1980	1985	1990	1995	2000	2000:1980
美元/MB	19 200	2 900	320	256	100	190
访问时间（ns）	300	150	35	15	3	100

（a）SRAM 趋势

度量标准	1980	1985	1990	1995	2000	2000:1980
美元/MB	8 000	880	100	30	1	8 000
访问时间（ns）	375	200	100	70	60	6
典型的大小（MB）	0.064	0.256	4	16	64	1 000

（b）DRAM 趋势

度量标准	1980	1985	1990	1995	2000	2000:1980
美元/MB	500	100	8	0.30	0.01	50 000
访问时间（ms）	87	75	28	10	8	11
典型的大小（MB）	1	10	160	1 000	20 000	20 000

（c）磁盘趋势

度量标准	1980	1985	1990	1995	2000	2000.1980
Intel CPU	8 080	80 286	80 386	Pentium	P-III	—
CPU 时钟频率（MHz）	1	6	20	150	600	600
CPU 时钟周期（ns）	1 000	166	50	6	1.6	600

（d）CPU 趋势

图 6.15 存储和处理器技术发展趋势

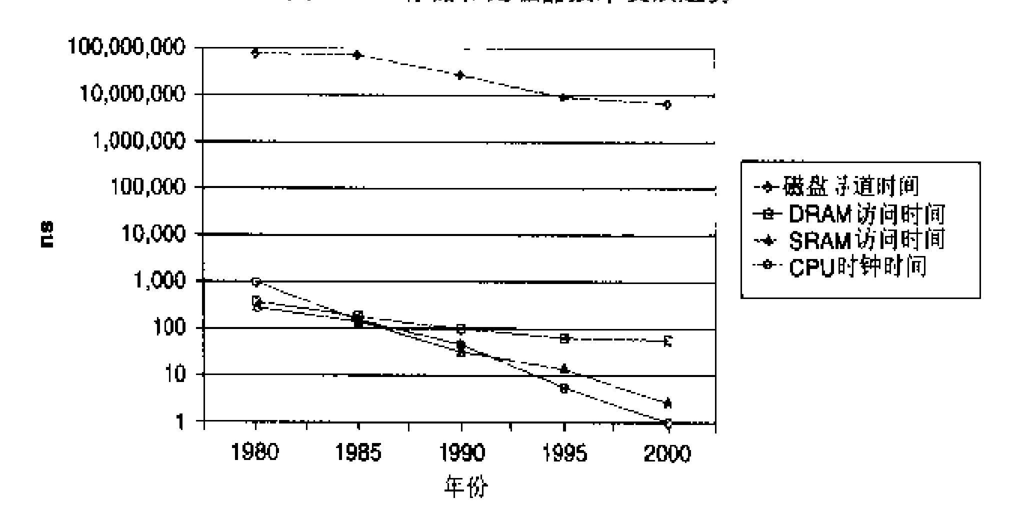


图 6.16 DRAM、磁盘和 CPU 速度之间逐渐增大的差距

正如我们将在6.4节中看到的那样，现代计算机频繁地使用基于SRAM的高速缓存，以弥补处理器-存储器之间的差距。这种方法行之有效是因为应用程序的一个称为*局部性*（locality）的基本属性，接下来我们就讨论这个问题。

6.2 局部性

一个编写良好的计算机程序倾向于展示出良好的*局部性*（locality）。也就是，它们倾向于引用的数据项邻近于其他最近引用过的数据项，或者邻近于最近自我引用过的数据项。这种倾向性，被称为*局部性原理*（principle of locality），是一个持久的概念，对硬件和软件系统的设计都有着极大的影响。

局部性通常有两种形式：*时间局部性*（temporal locality）和*空间局部性*（spatial locality）。在一个具有良好时间局部性的程序中，被引用过一次的存储器位置很可能在不远的将来再被多次引用。在一个具有良好空间局部性的程序中，如果一个存储器位置被引用了一次，那么程序很可能在不远的将来引用附近的一个存储器位置。

程序员应该理解局部性原理，因为一般而言，*有良好局部性的程序比局部性差的程序运行得更快*。现代计算机系统的各个层次，从硬件到操作系统、到应用程序，它们的设计都利用了局部性。在硬件层，局部性原理允许计算机设计者通过引入称为*高速缓存存储器*的小而快速的存储器来保存最近被引用的指令和数据项，从而提高对主存的访问速度。在操作系统级，局部性原理允许系统使用主存作为虚拟地址空间最近被引用块的高速缓存。类似地，操作系统用主存来缓存磁

盘文件系统中最近被使用的磁盘块。局部性原理在应用程序的设计中也扮演着重要的角色。例如，Web 浏览器将最近被引用的文档放在本地磁盘上，利用的就是时间局部性。大量的 Web 服务器将最近被请求的文档放在前端磁盘高速缓存中，这些缓存能满足对这些文档的请求，而不需要服务器的任何干涉。

6.2.1 对程序数据引用的局部性

考虑图 6.17（a）中的简单函数，它对一个向量的所有元素求和。这个程序有良好的局部性吗？为了回答这个问题，我们来看看每个变量的引用模式。在这个例子中，变量 sum 在每次循环迭代中被引用一次，因此，对于 sum 来说，有好的局部性。另一方面，因为 sum 是标量，对于 sum 来说，没有空间局部性。

正如我们在图 6.17（b）中看到的，向量 v 的元素是被顺序读取的，一个接一个，按照它们存储在存储器中的顺序（为了方便，我们假设数组是从地址 0 开始的）。因此，对于变量 v，函数有很好的空间局部性，但是时间局部性很差，因为每个向量元素只被访问一次。因为对于循环体中的每个变量，这个函数要么有好的空间局部性，要么有好的时间局部性，所以我们可以断定 sumvec 函数有良好的局部性。

```
1   int sumvec(int v[N])
2   {
3       int i, sum = 0;
4
5       for (i = 0; i < N; i++)
6           sum += v[i];
7       return sum;
8   }
```

(a)

地址内容	0 v_0	4 v_1	8 v_2	12 v_3	16 v_4	20 v_5	24 v_6	28 v_7
访问顺序	1	2	3	4	5	6	7	8

(b)

图 6.17 (a) 一个具有良好局部性的程序；(b) 向量 v 的引用模式 (N = 8)

注意如何按照向量元素存储在存储器中的顺序来访问它们。

我们说像 sumvec 这样顺序访问一个向量每个元素的函数，具有步长为 1 的引用模式（stride-1 reference pattern）（相对于元素的大小）。访问一个连续的向量的每第 k 个元素，就被称为步长为 k 的引用模式（stride-k reference pattern）。步长为 1 的引用模式是程序中空间局部性常见和重要的来源。一般而言，随着步长的增加，空间局部性下降。

对于引用多维数组的程序来说，步长也是一个很重要的问题。考虑图 6.18（a）中的函数 sumarrayrows，它对一个二维数组的元素求和。双重循环按照行优先顺序（row-major order）读数组的元素。也就是，内层循环读第一行的元素，依此类推。函数 sumarrayrows 具有良好的空间局部性，因为它按照数组被存储的行优先顺序来访问这个数组，如图 6.18（b）所示。其结果是得到一个很

好的步长为1的引用模式和良好的空间局部性。

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6          for (j = 0; j < N; j++)
7             sum += a[i][j];
8       return sum;
9   }
```

(a)

地址内容	0 a_{00}	4 a_{01}	8 a_{02}	12 a_{10}	16 a_{11}	20 a_{12}
访问顺序	1	2	3	4	5	6

(b)

图 6.18 (a) 另一个具有良好局部性的程序；(b) 数组a的引用模式（M=2，N=3）
有良好的空间局部性，是因为数组是按照与它存储在存储器中一样的行优先顺序来被访问的。

一些看上去很小的对程序的改动能够对它的局部性有很大的影响。例如，图 6.19（a）中的函数 sumarraycols 计算和图 6.18（a）中函数 sumarrayrows 一样的结果。惟一的区别是我们交换了 i 和 j 的循环。这样交换循环对它的局部性有何影响？

糟糕的空间局部性损害了函数 sumarraycols，因为它按照列来扫描数组，而不是按照行。因为 C 数组在存储器中是按照行来存放的，结果就得到步长为(N × sizeof(int))的引用模式，如图 6.19（b）所示。

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

(a)

地址内容	0 a_{00}	4 a_{01}	8 a_{02}	12 a_{10}	16 a_{11}	20 a_{12}
访问顺序	1	3	5	2	4	6

(b)

图 6.19 (a) 一个空间局部性很差的程序；(b) 数组a的引用模式（M=2，N=3）
函数的空间局部性很差，这是因为它使用步长为(N × sizeof(int))的引用模式来扫描存储器。

6.2.2 取指令的局部性

因为程序指令是存放在存储器中的，CPU 必须取出（读出）这些指令，所以我们也能够评价一个程序关于取指令的局部性。例如，图 6.17 中 for 循环体里的指令是按照连续的存储器顺序执行的，因此循环有良好的空间局部性。因为循环体会被执行多次，所以它也有很好的时间局部性。

代码区别于程序数据的一个重要属性是在运行时它是不能被修改的。当程序正在执行时，CPU 只从存储器中读出它的指令。CPU 决不会重写或修改这些指令。

6.2.3 局部性小结

在这一节中，我们介绍了局部性的基本思想，还给出了一些量化评价一个程序中局部性的简单原则：

- 重复引用同一个变量的程序有良好的时间局部性。
- 对于具有步长为 k 的引用模式的程序，步长越小，空间局部性越好。具有步长为 1 的引用模式的程序有很好的空间局部性。在存储器中以大步长跳来跳去的程序空间局部性会很差。
- 对于取指令来说，循环有好的时间和空间局部性。循环体越小，循环迭代次数越多，局部性越好。

到本章后面，在我们学习了高速缓存存储器以及它们是如何工作的之后，我们会向你展示如何用高速缓存命中率和不命中率来量化局部性的概念。你还会弄明白为什么有良好局部性的程序通常比局部性差的程序运行得更快。尽管如此，了解如何看一眼源代码就能获得对程序中局部性的高级感觉，是程序员要掌握的一项有用而且重要的技能。

练习题 6.4

改变下面函数中循环的顺序，使得它以步长为 1 的引用模式扫描三维数组 a：

```
1   int sumarray3d(int a[N][N][N])
2   {
3       int i, j, k, sum = 0;
4
5       for (i = 0; i < N; i++) {
6           for (j = 0; j < N; j++) {
7               for (k = 0; k < N; k++) {
8                   sum += a[k][i][j];
9               }
10          }
11      }
12      return sum;
13  }
```

练习题 6.5

图 6.20 中的三个函数，以不同的空间局部性程度，执行相同的操作。请对这些函数的空间局部性进行排序。解释你是如何得到你的排序结果的。

```
1   #define N 1000
2
3   typedef struct {
4       int vel[3];
5       int acc[3];
6   } point;
7
8   point p[N];
```

(a) structs 数组

```
1   void clear1(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++)
7               p[i].vel[j] = 0;
8           for (j = 0; j < 3; j++)
9               p[i].acc[j] = 0;
10      }
11  }
```

(b) clear1 函数

```
1   void clear2(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++) {
7               p[i].vel[j] = 0;
8               p[i].acc[j] = 0;
9           }
10      }
11  }
```

(c) clear2 函数

```
1   void clear3(point *p, int n)
2   {
3       int i, j;
4
5       for (j = 0; j < 3; j++) {
6           for (i = 0; i < n; i++)
7               p[i].vel[j] = 0;
8           for (i = 0; i < n; i++)
9               p[i].acc[j] = 0;
10      }
11  }
```

(d) clear3 函数

图 6.20 练习题 6.5 的代码示例

6.3 存储器层次结构

6.1 节和 6.2 节描述了存储技术和计算机软件的一些基本的和持久的属性：

- 不同存储设备的访问时间差异很大。速度较快的设备每字节的成本要比速度较慢的设备高，而且容量较小。CPU 和主存之间的速度差距在增大。
- 一个编写良好的程序倾向于展示出良好的局部性。

计算技术中一个喜人的巧合是，硬件和软件的这些基本属性互相补充得很完美。它们这种相互补充的性质使人想到一种组织存储器系统的方法，称为*存储器层次结构*（memory hierarchy），所有的现代计算机系统中都使用了这种方法。图 6.21 展示了一个典型的存储器层次结构。

一般而言，从高层往底层走，存储设备变得更慢，更便宜和更大。在最高层（L0），是少量的快速 CPU 寄存器，CPU 可以在一个时钟周期内访问它们。接下来是一个或多个小型或中型的基于 SRAM 的高速缓存存储器，可以在几个 CPU 时钟周期内访问它们。然后是一个大的基于 DRAM 的主存，可以在几十或几百个时钟周期内访问它们。接下来是慢速但是容量很大的本地磁盘。最

后，有些系统甚至包括了一层附加的远程服务器上的磁盘，要通过网络来访问它们。例如，像安德鲁文件系统（AFS）或者网络文件系统（NFS）这样的分布式文件系统，允许程序访问存储在远程的网络服务器上的文件。类似地，万维网允许程序访问存储在世界上任何地方的 Web 服务器上的远程文件。

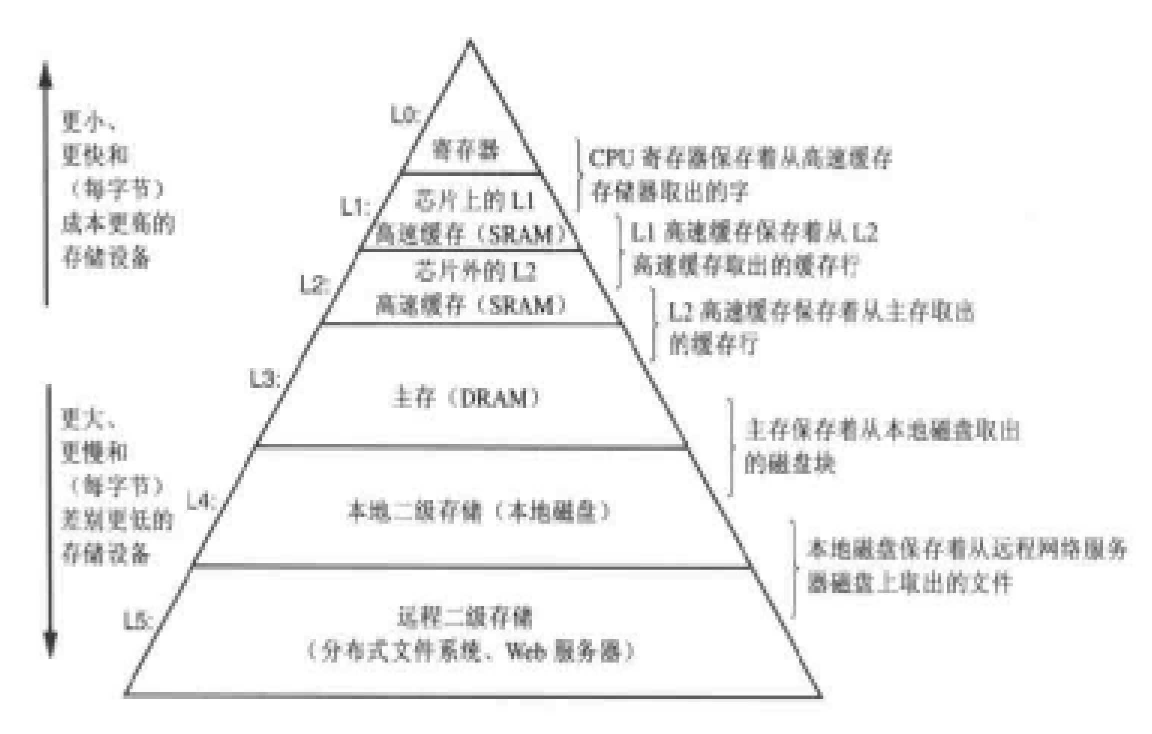


图 6.21 存储器层次结构

旁注：其他的存储器层次结构

我们向你展示了一个存储器层次结构的示例，但是其他的组合也是可能的，而且确实也很常见。例如，许多地方将本地磁盘备份到存档的磁带上。其中有些地方，在需要时是由人来手工地装好磁带的，而其中还有些地方是由磁带机器人自动地完成这项任务的。无论在哪种情况中，磁带都是存储器层次结构中的一层，在本地磁盘那一层下面，那些一般的原则也同样适用于它。磁带每字节比磁盘更便宜，它允许人们将本地磁盘的多个快照存档，但是磁带的访问时间要比磁盘的更长。

6.3.1 在存储器层次结构中的缓存

一般而言，高速缓存（cache，读作 "cash"）是一个小而快速的存储设备，它作为存储在更大、也更慢的设备中的数据对象的缓冲区域。使用高速缓存的过程被称为缓存（caching，读作"cashing"）。

存储器层次结构的中心思想是，对于每个 k，位于 k 层的更快更小的存储设备作为位于 k+1 层的更大更慢的存储设备的缓存。换句话说，层次结构中的每一层都缓存来自较低一层的数据对象。例如，本地磁盘作为通过网络从远程磁盘取出的文件（例如 Web 页面）的缓存，主存作为本地磁盘上数据的缓存，依此类推，直到最小的缓存——CPU 寄存器集合。

图 6.22 展示了存储器层次结构中缓存的一般性概念。第 k+1 层的存储器被划分成连续的数据对象组块（chunks），称为块（blocks）。每个块都有一个惟一的地址或名字，使之区别于其他的块。块可以是固定大小的（通常是这样的），也可以是可变大小的（例如，存储在 Web 服务器上的远程 HTML

文件）。例如，图 6.22 中第 k+1 层存储器被划分成 16 个大小固定的块，编号为 0～15。

类似地，第 k 层的存储器被划分成较小的块的集合，每个块的大小与 k+1 层的块的大小一样。在任何时刻，第 k 层的缓存包含第 k+1 层块的一个子集的拷贝。例如，在图 6.22 中，第 k 层的缓存有 4 个块的空间，当前包含块 4、9、14 和 3 的拷贝。

数据总是以块大小为传送单元（transfer unit）在第 k 层和第 k+1 层之间来回拷贝的。虽然在层次结构中任何一对相邻的层次之间块大小是固定的，但是其他的层次对之间可以有不同的块大小。例如，在图 6.21 中，L1 和 L0 之间的传送通常使用的是 1 个字的块。L2 和 L1 之间（以及 L3 和 L2 之间）的传送通常使用的是 4～8 个字的块。而 L4 和 L3 之间的传送用的是大小为几百或几千字节的块。一般而言，层次结构中较低层（离 CPU 较远）的设备的访问时间较长，因此为了补偿这些较长的访问时间，倾向于使用较大的块。

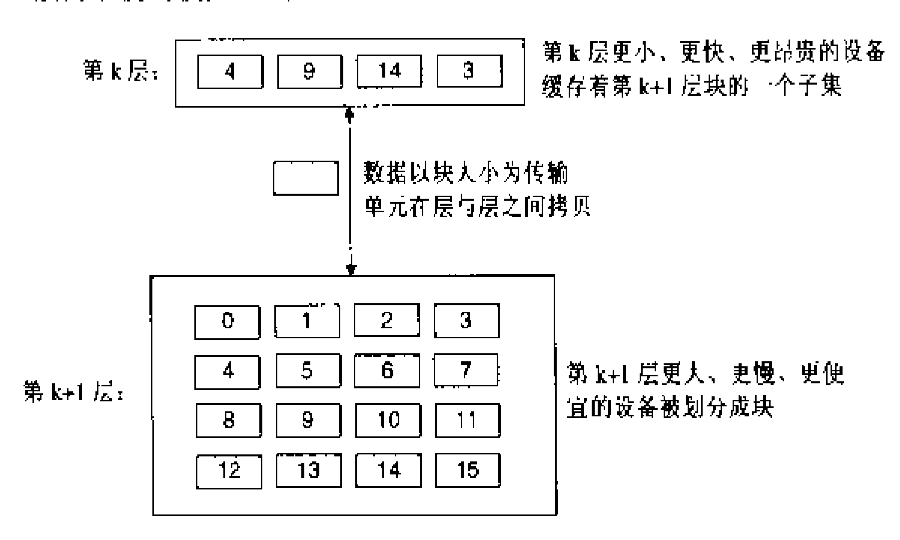


图 6.22　存储器层次结构中一个基本的缓存原理

缓存命中

当程序需要第 k+1 层的某个数据对象 d 时，它首先在当前存储在第 k 层的一个块中查找 d。如果 d 刚好缓存在第 k 层中，那么就是我们所说的*缓存命中*（cache hit）。该程序直接从第 k 层读取 d，根据存储器层次结构的性质，这要比从第 k+1 层读取 d 更快。例如，一个有良好时间局部性的程序可以从块 14 中读出一个数据对象，得到一个对第 k 层的缓存命中。

缓存不命中

另一方面，如果第 k 层中没有缓存数据对象 d，那么就是我们所说的*缓存不命中*（cache miss）。当发生缓存不命中时，第 k 层的缓存从第 k+1 层缓存中取出包含 d 的那个块，如果第 k 层的缓存已经满了的话，可能就会覆盖现存的一个块。

覆盖一个现存的块的过程被称为替换（replacing）或驱逐（evicting）这个块。被驱逐的这个块有时也被称为*牺牲块*（victim block）。决定该替换哪个块是由缓存的替换策略来控制的。例如，一个具有随机替换策略的缓存会随机选择一个牺牲块。一个具有最近最少被使用（LRU）替换策略的缓存会选择那个最后被访问的时间距现在最远的块。

在第 k 层缓存从第 k+1 层取出那个块之后，程序就能像前面一样从第 k 层读出 d 了。例如，在图 6.22 中，在第 k 层中读块 12 中的一个数据对象，会导致一个缓存不命中，因为块 12 当前不在第

k 层缓存中。一旦把块 12 从第 k+1 层拷贝到第 k 层之后，它就会保持在那里，等待稍后的访问。

缓存不命中的种类

区分不同种类的缓存不命中有时候是很有帮助的。如果第 k 层的缓存是空的，那么对任何数据对象的访问都会不命中。一个空的缓存有时被称为*冷缓存*（cold cache），此类不命中被称为*强制性不命中*（compulsory miss）或*冷不命中*（cold miss）。冷不命中很重要，因为它们通常是短暂的事件，不会在稳定状态中出现，稳定状态指的就是在反复的存储器访问已经将缓存变暖（warmed up）了之后。

只要发生了不命中，第 k 层的缓存就必须执行某个*替换策略*，确定把它从第 k+1 层中取出的块放在哪里。最灵活的替换策略是允许来自第 k+1 层的任何块放在第 k 层的任何块中。对于存储器层次结构中高层的缓存（靠近 CPU），它们是用硬件来实现的，而且速度是最优的，这个策略实现起来通常很昂贵，因为随机地放置块，定位起来代价很高。

因此，硬件缓存通常使用的是更严格的放置策略，这个策略将第 k+1 层的某个块限制放置在第 k 层块的一个小的子集中（有时只是一个块）。例如，在图 6.22 中，我们可以确定第 k+1 层的块 i 必须放置在第 k 层的块（i mod 4）中。例如，第 k+1 层的块 0、4、8 和 12 会映射到第 k 层的块 0，块 1、5、9 和 13 会映射到块 1，依此类推。注意，图 6.22 中我们的示例缓存使用的就是这个策略。

这种限制性的放置策略会引起一种不命中，称为*冲突不命中*（conflict miss），在这种情况中，缓存足够大，能够保存被引用的数据对象，但是因为这些对象会映射到同一个缓存块，缓存会一直不命中。例如，在图 6.22 中，如果程序请求块 0，然后块 8，然后块 0，然后块 8，依此类推，在第 k 层的缓存中，对这两个块的每次引用都会不命中，即使是这个缓存总共可以容纳 4 个块。

程序通常是按照一系列阶段（例如，循环）来运行的，每个阶段访问缓存块的某个相对稳定不变的集合。例如，一个嵌套的循环可能会反复地访问同一个数组的元素。这个块的集合被称为这个阶段的*工作集*（working set）。当工作集的大小超过缓存的大小时，缓存会经历*容量不命中*（capacity miss）。换句话说，缓存就是太小了，不能处理这个工作集。

高速缓存管理

正如我们提到过的，存储器层次结构的本质是，每一层存储设备都是较低一层的缓存。在每一层上，某种形式的逻辑必须*管理*缓存。这里，我们的意思是指某个东西要将缓存划分成块，在不同的层之间传送块，判定是命中还是不命中，并处理它们。管理缓存的逻辑可以是硬件、软件，或是两者的结合。

例如，编译器管理寄存器文件，缓存层次结构的最高层。它决定当发生不命中时何时发射加载，以及确定哪个寄存器来存放数据。L1 和 L2 层的缓存完全是由内置在缓存中的硬件逻辑来管理的。在一个有虚拟存储器的系统中，DRAM 主存作为存储在磁盘上的数据块的缓存，是由操作系统软件和 CPU 上的地址翻译硬件共同管理的。对于一个具有像 AFS 这样的分布式文件系统的机器来说，本地磁盘作为缓存，它是由运行在本地机器上的 AFS 客户端进程管理的。在大多数时候，缓存都是自动运行的，不需要程序采取特殊的或显式的行动。

6.3.2 存储器层次结构概念小结

概括来说，存储器层次结构行之有效，是因为较慢的存储设备比较快的存储设备更便宜，还因为程序倾向于展示局部性：

- 利用时间局部性：根据时间局部性，同一数据对象可能会被多次使用。一旦一个数据对象在第一次不命中时被拷贝到缓存中，我们就会期望后面对该目标有一系列的访问命中。因为缓存比低一层的存储设备更快，对后面的命中的服务会比最开始的不命中快很多。
- 利用空间局部性：块通常包含有多个数据对象。根据空间局部性，我们会期望后面对该块中其他对象的访问能够补偿不命中后拷贝该块的花费。

现代系统中到处都使用了缓存。正如从图6.23中能够看到的那样，CPU芯片、操作系统、分布式文件系统中和万维网上都使用了缓存。各种各样硬件和软件的组合构成和管理着缓存。注意，图6.23中有大量我们还未涉及到的术语和缩写。在此我们包括这些术语和缩写是为了说明常见的缓存是什么样子的。

类型	缓存的内容	被缓存在何处	执行时间(周期数)	由谁管理
CPU寄存器	4字节字	芯片上的CPU寄存器	0	编译器
TLB	地址翻译	芯片上的TLB	0	硬件MMU
L1高速缓存	32字节块	芯片上的L1高速缓存	1	硬件
L2高速缓存	32字节块	芯片外的L2高速缓存	10	硬件
虚拟存储器	4-KB页	主存	100	硬件 + OS
缓冲区高速缓存	部分文件	主存	100	OS
网络缓冲区	部分文件	本地磁盘	10 000 000	AFS/NFS客户
浏览器高速缓存	Web页	本地磁盘	10 000 000	Web浏览器
Web高速缓存	Web页	远程服务器磁盘	1 000 000 000	Web代理服务器

图6.23 缓存在现代计算机系统中无处不在

TLB：翻译后备缓冲器（Translation Lookaside Buffer）；MMU：存储器管理单元（Memory Management Unit）；OS：操作系统（Operating System）；AFS：安德鲁文件系统（Andrew File System）；NFS：网络文件系统（Network File System）。

6.4 高速缓存存储器

早期计算机系统的存储器层次结构只有三层：CPU寄存器、主DRAM存储器和磁盘存储设备。不过，由于CPU和主存之间逐渐增大的差距，系统设计者被迫在CPU寄存器文件和主存之间插入了一个小的SRAM存储器，称为L1高速缓存（一级缓存）。在现代系统中，L1高速缓存位于CPU芯片上（也就是，它是芯片上的高速缓存），如图6.24所示。L1高速缓存的访问速度几乎和寄存器一样快，典型地是1个或2个时钟周期。

随着CPU和主存之间的性能差距不断增大，系统设计者在L1高速缓存和主存之间又插入了一个高速缓存，称为L2高速缓存，可以在几个时钟周期内访问到它。可以将L2高速缓存连接到存储器总线，或者连接到它自己的高速缓存总线（cache bus），如图6.24所示。有些高性能系统，例如那些基于Alpha 21164的系统，甚至于在存储器总线上还有一层高速缓存，称为L3高速缓存，在层次结构中位于L2高速缓存和主存之间。虽然在安排上有相当多的种类，但是一般的原则都是一样的。

6.4.1 通用的高速缓存存储器结构

考虑一个计算机系统，其中每个存储器地址有m位，形成$M = 2^m$个不同的地址。如图6.25（a）所示，这样一个机器的高速缓存被组织成一个$S = 2^s$个高速缓存组（cache set）的数组。每个组包含

E个高速缓存行（cache line）。每个行是由一个 $B=2^b$ 字节的数据块（block）组成的，一个有效位（valid bit）指明这个行包含的数据是否有意义，还有 $t=m-(b+s)$ 个标记位（tag bit）（是当前块的存储器地址的位子集），它们惟一地标识存储在这个高速缓存行中的块。

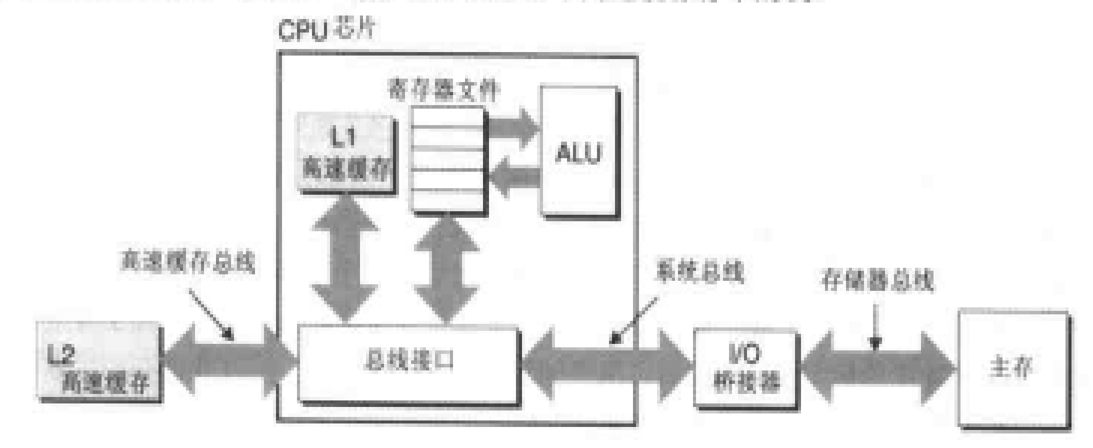


图 6.24 基于 L1 和 L2 高速缓存的典型总线结构

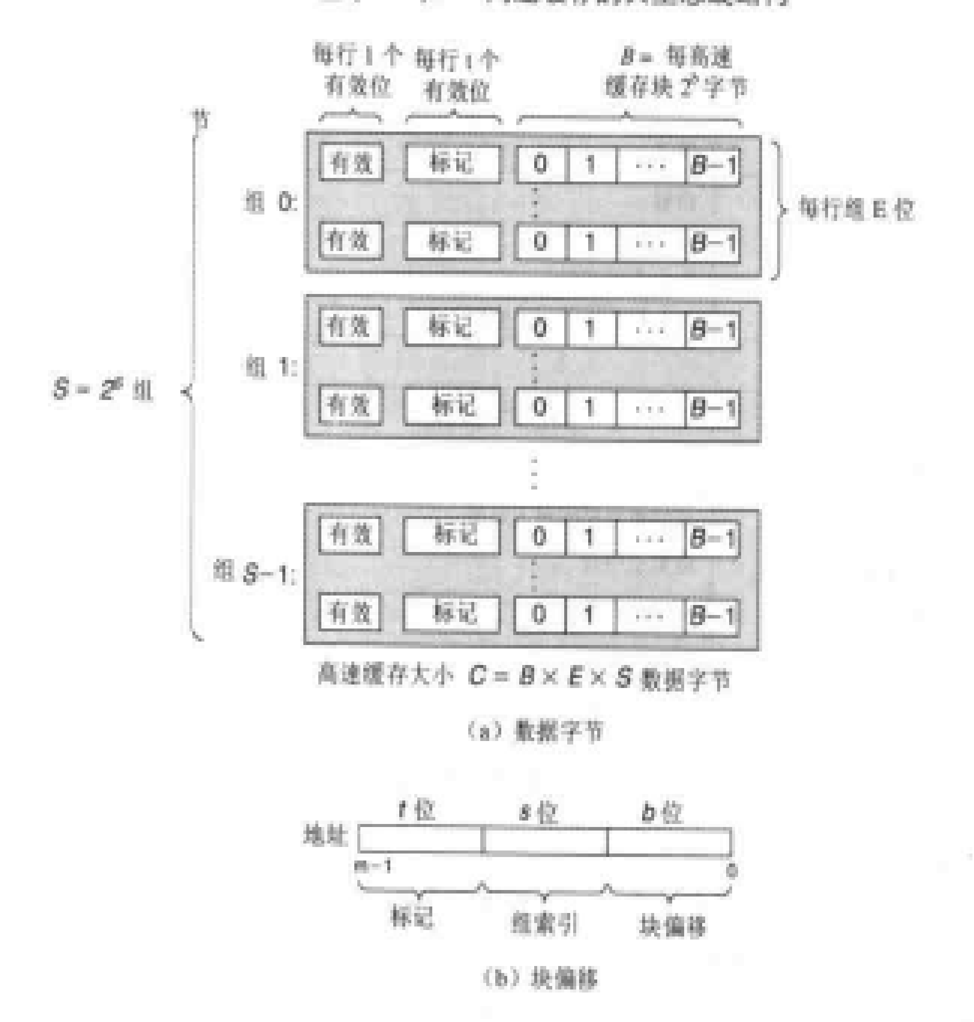


图 6.25 高速缓存（S,E,B,m）的通用结构

（a）高速缓存是一个关于组的数组。每个组包含一个或多个行，每个行包含一个有效位，一些标记位，以及一个数据块；（b）高速缓存的结构将 m 个地址位划分成了 t 个标记位、s 个组索引位和 b 个块偏移位。

一般而言，高速缓存的结构可以用元组（S,E,B,m）来描述。高速缓存的大小（或容量）C 指的是所有块的大小的和。标记位和有效位不包括在内。因此，$C = S \times E \times B$。

当一条加载指令指示 CPU 从主存地址 A 中读一个字时，它将地址 A 发送到高速缓存。如果高速缓存正保存着地址 A 处那个字的拷贝，它就立即将那个字发回给 CPU。那么高速缓存如何知道它是否包含地址 A 处那个字的拷贝的呢？高速缓存的结构使得它能通过简单地检查地址位，发现所请求的字，类似于使用极其简单的哈希函数的哈希表。下面就是它是如何工作的：

参数 S 和 B 将 m 个地址位分为了三个字段，如图 6.25（b）所示。A 中 s 个组索引位是一个到 S 个组的数组索引。第一个组是组 0，第二个组是组 1，依此类推。当作为一个无符号整数解释时，组索引位告诉我们这个字必须存储在哪个组中。一旦我们知道了这个字必须放在哪个组中，A 中的 t 个标记位就告诉我们这个组中的哪一行（如果有的话）包含这个字。当且仅当设置了有效位并且该行的标记位与地址 A 中的标记位相匹配时，组中的这一行包含这个字。一旦我们在由组索引标识的组中定位了由标号所标识的行，那么 b 个块偏移位给出了在 B 个字节的数据块中的字偏移。

你可能已经注意到了，对高速缓存的描述使用了很多符号。图 6.26 对这些符号做了个小结，供你参考。

基本参数	
参数	**描述**
$S = 2^s$	组数
E	每个组的行数
$B = 2^b$	块大小（字节）
$m = \log_2(M)$	（主存）物理地址位数

衍生出来的量	
参数	**描述**
$M = 2^m$	存储器地址的最大数量
$s = \log_2(S)$	组索引位数
$b = \log_2(B)$	块偏移位数
$t = m - (s + b)$	标记位数
$C = B \times E \times S$	不包括像有效位和标记位这样开销的高速缓存大小（字节）

图 6.26 高速缓存参数小结

练习题 6.6

下表给出了几个不同高速缓存的参数。确定每个高速缓存的高速缓存组数（S）、标记位数（t）、组索引位数（s）以及块偏移位数（b）。

高速缓存	m	C	B	E	S	t	s	b
1.	32	1024	4	1				
2.	32	1024	8	4				
3.	32	1024	32	32				

6.4.2 直接映射高速缓存

根据E（每个组的高速缓存行数），高速缓存被分为不同的类。每个组只有一行（$E = 1$）的高速缓存被称为直接映射高速缓存（direct-mapped cache）（参见图6.27）。直接映射高速缓存是最容易实现和理解的，所以我们会以它为例来说明一些关于高速缓存是如何工作的一般概念。

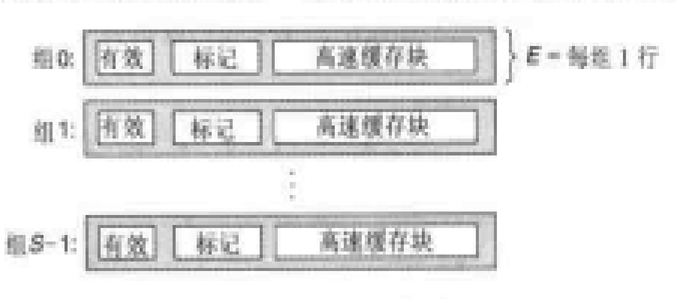


图 6.27 直接映射高速缓存（$E=1$）

每个组只有一行。

假设我们有这样一个系统，它有一个CPU、一个寄存器文件、一个L1高速缓存和一个主存。当CPU执行一条读存储器字w的指令，它向L1高速缓存请求这个字。如果L1高速缓存有w的一个缓存的拷贝，那么就得到L1高速缓存命中，高速缓存会很快抽取出w，并将它返回给CPU。否则就是高速缓存不命中，当L1高速缓存向主存请求包含w的块的一个拷贝时，CPU必须等待。当被请求的块最终从存储器到达时，L1高速缓存将这个块存放在它的一个高速缓存行里，从被存储的块中抽取出字w，然后将它返回给CPU。高速缓存确定一个请求是否命中，然后抽取出被请求的字的过程，分为三步：①组选择；②行匹配；③字抽取。

直接映射高速缓存中的组选择

在这一步中，高速缓存从w的地址中间抽取出s个组索引位。这些位被解释成一个对应于一个组号的无符号整数。换句话来说，如果我们把高速缓存看成是一个关于组的一维数组，那么这些组索引位就是一个到这个数组的索引。图6.28展示了直接映射高速缓存的组选择是如何工作的。在这个例子中，组索引位00001_2被解释为一个选择组1的整数索引。

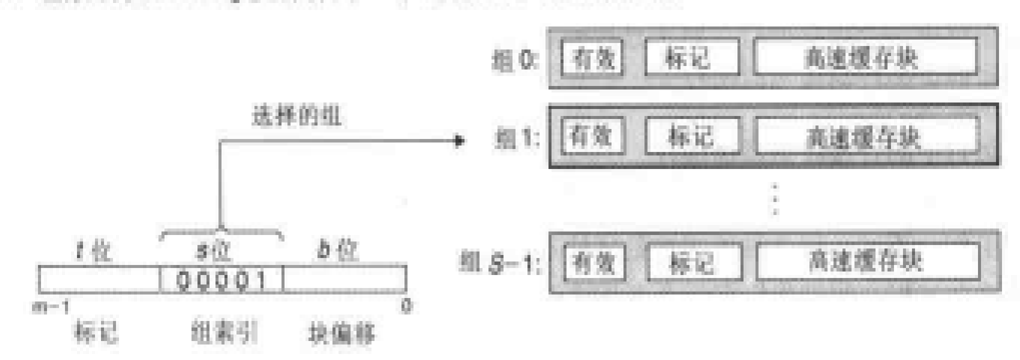


图 6.28 直接映射高速缓存中的组选择

直接映射高速缓存中的行匹配

既然在上一步中我们已经选择了某个组i，接下来的一步就要确定是否有字w的一个拷贝存储在组i包含的一个高速缓存行中。在直接映射高速缓存中，这很容易，而且很快，这是因为每个组只有一行。当且仅当设置了有效位，而且高速缓存行中的标记与w的地址中的标记相匹配时，这一

行中包含 w 的一个拷贝。

图 6.29 展示了直接映射高速缓存中行匹配是如何工作的。这个行的有效位设置了，所以我们知道标记中的位和块是有意义的。因为这个高速缓存行中的标记位与地址中的标记位相匹配，所以我们知道我们想要的那个字的一个拷贝确实存储在这个行中。换句话说，我们得到一个缓存命中。另一方面，如果有效位没有设置，或者标记不相匹配，那么我们就得到一个缓存不命中。

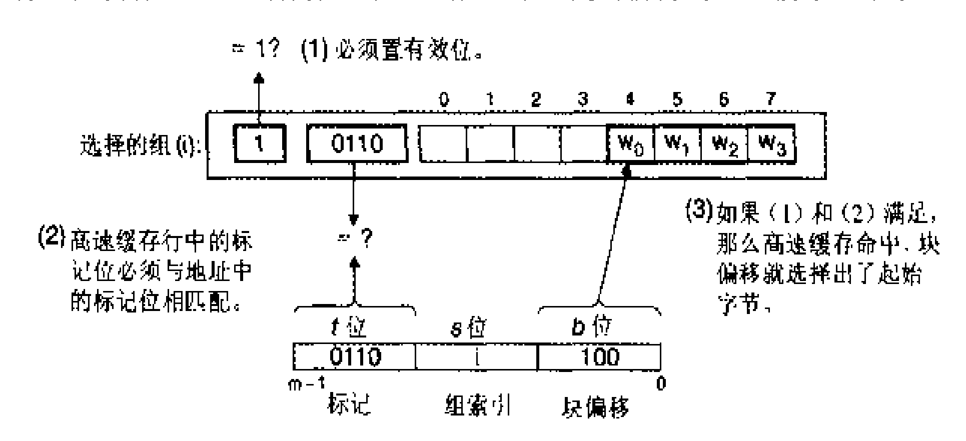


图 6.29 直接映射高速缓存中的行匹配和字选择

在高速缓存块中，w_0 表示字 w 的低位字节，w_1 是下一个字节，依此类推。

直接映射高速缓存中的字选择

一旦命中，我们知道 w 就在这个块中的某个地方。最后一步确定所需要的字在块中是从哪里开始的。如图 6.29 所示，块偏移位向我们提供了所需要的字的第一个字节的偏移。就像我们把高速缓存看成一个行的数组一样，我们把块看成一个字节的数组，而字节偏移是到这个数组的一个索引。在这个示例中，块偏移位是 100_2，它表明 w 的拷贝是从块中的字节 4 开始的（我们假设字长为 4 字节）。

直接映射高速缓存中不命中时的行替换

如果高速缓存不命中，那么它需要从存储器层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。一般而言，如果组中都是有效高速缓存行了，那么必须要驱逐出一个现存的行。对于直接映射高速缓存来说，每个组只包含有一行，替换策略非常简单：用新取出的行替换当前的行。

综合：运行中的直接映射高速缓存

高速缓存用来选择组和标识行的机制极其简单。必须要这样，因为硬件必须在几个纳秒的时间内完成这些工作。不过，用这种方式来处理位对我们人来说是很令人困惑的。一个具体的例子能帮助解释清楚这个过程。假设我们有一个直接映射高速缓存，描述如下

$$(S, E, B, m) = (4, 1, 2, 4)$$

换句话说，高速缓存有四个组，每个组一行，每个块 2 个字节，而地址是 4 位的。我们还假设每个字都是单字节的。当然这样一些假设完全是不现实的，但是它们能使示例保持简单。

如果你初学高速缓存，列举出整个地址空间并划分好位，是很有帮助的，就像我们在图 6.30 对我们 4 位的示例所做的那样。关于这个列举出的空间，有一些有趣的事情值得注意：

- 标记位和索引位连起来惟一地标识了存储器中的每个块。例如，块 0 是由地址 0 和 1 组成

的，块 1 是由地址 2 和 3 组成的，块 2 是由地址 4 和 5 组成的，依此类推。

- 因为有 8 个存储器块，但是只有 4 个高速缓存组，多个块映射到同一个高速缓存组（也就是，它们有相同的组索引）。例如，块 0 和 4 都映射到组 0，块 1 和 5 都映射到组 1，等等。
- 映射到同一个高速缓存组的块由标识位惟一地标识。例如，块 0 的标识位为 0，而块 4 的标识位为 1，块 1 的标识位为 0，而块 5 的标识位为 1。

	地址位			
地址（十进制）	标记位（t=1）	索引位（s=2）	偏移位（b=1）	块号（十进制）
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

图 6.30 示例直接映射高速缓存的 4 位地址空间

让我们来模拟一下当 CPU 执行一系列读的时候，高速缓存的执行情况。记住对于这个示例，我们假设 CPU 读 1 字节的字。虽然这种手工的模拟很乏味，你可能想要跳过它，但是根据我们的经验，在学生们经历过高速缓存是如何工作的之前，他们是不能真正理解的。

初始时，缓存是空的（也就是，每个有效位都是 0）：

组	有效位	标记位	块[0]	块[1]
0	0			
1	0			
2	0			
3	0			

表中的每一行都代表一个高速缓存行。第一列表明该行所属的组，但是请记住提供这个位只是为了方便，实际上它并不真是缓存的一部分。后面四列代表每个高速缓存行的实际的位。现在，让

我们来看看当CPU执行一系列读时，都发生了什么：

1. **地址0的字。**因为组0的有效位是0，是缓存不命中。高速缓存从存储器（或低一层的高速缓存）取出块0，并把这个块存储在组0中。然后，高速缓存返回新取出的高速缓存行的块[0]的m[0]（存储器位置0的内容）。

组	有效位	标记位	块[0]	块[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. **读地址1的字。**这次会缓存命中。高速缓存立即从高速缓存行的块[1]中返回m[1]。高速缓存的状态没有变化。

3. **读地址13的字。**由于组2中的高速缓存行不是有效的，所以有缓存不命中。高速缓存把块6加载到组2中，然后从新的高速缓存行的块[1]中返回m[13]。

组	有效位	标记位	块[0]	块[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. **读地址8的字。**这会发生缓存不命中。组0中的高速缓存行确实是有效的，但是标记不匹配。高速缓存将块4加载到组0中（替换读地址0时读入的那一行），然后从新的高速缓存行的块[0]中返回m[8]。

组	有效位	标记位	块[0]	块[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. **读地址0的字。**又会发生缓存不命中，因为在前面引用地址8时，我们刚好替换了块0。这就是冲突不命中的一个例子，也就是我们有足够的高速缓存空间，但是交替地引用映射到同一个组的块。

组	有效位	标记位	块[0]	块[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

直接映射高速缓存中的冲突不命中

冲突不命中在真实的程序中很常见，会导致令人困惑的性能问题。当程序访问大小为2的幂的

数组时，直接映射高速缓存中通常会发生冲突不命中。例如，考虑一个计算两个向量点积的函数：

```
1    float dotprod(float x[8], float y[8])
2    {
3        float sum = 0.0;
4        int i;
5
6        for (i = 0; i < 8; i++)
7            sum += x[i] * y[i];
8        return sum;
9    }
```

对于 x 和 y 来说，这个函数有良好的局部性，因此我们期望它的命中率会比较高。不幸的是，并不总是如此。

假设浮点数是 4 个字节，x 被加载到从地址 0 开始的 32 字节连续存储器中，而 y 紧跟在 x 之后，从地址 32 开始。为了简便，假设一个块是 16 个字节（足够容纳 4 个浮点数），高速缓存由两个组组成，高速缓存的整个大小为 32 字节。我们会假设变量 sum 实际上存放在一个 CPU 寄存器中，因此不需要存储器引用。根据这些假设每个 x[i]和 y[i]会映射到相同的高速缓存组：

元素	地址	组索引	元素	地址	组索引
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

在运行时，循环的第一次迭代引用 x[0]，缓存不命中会导致包含 x[0] ~x[3]的块被加载到组 0。接下来是对 y[0]的引用，又一次缓存不命中，导致包含 y[0]~y[3]的块被拷贝到组 0，覆盖前一次引用拷贝进来的 x 的值。在下一次迭代中，对 x[1]的引用不命中，导致 x[0]~x[3]的块被加载回组 0，覆盖掉 y[0]~y[3]的块。因而现在我们就有了一个冲突不命中，而实际上后面每次对 x 和 y 的引用都会导致冲突不命中，我们就在 x 和 y 的块之间抖动（thrash）。术语“抖动”描述的是这样一种情况，其中高速缓存反复地加载和驱逐高速缓存块相同的组。

简要来说就是，即使程序有良好的空间局部性，而我们的高速缓存中也有足够的空间来存放 x[i] 和 y[i]的块，每次引用还是会导致冲突不命中，这是因为这些块被映射到了同一个高速缓存组。这种抖动导致速度下降 2 或 3 倍并不稀奇。另外，还要注意虽然我们的示例极其简单，但是对于更大、更现实的直接映射高速缓存来说，这个问题也是很真实的。

幸运的是，一旦程序员意识到了正在发生什么，就很容易修正抖动问题。一个很简单的方法是在每个数组的结尾放 B 字节的填充。例如，不是将 x 定义为 float x[8]，而是定义成 float x[12]。假设在存储器中 y 紧跟在 x 后面，我们有下面这样的从数组元素到组的映射：

元素	地址	组索引	元素	地址	组索引
x[0]	0	0	y[0]	48	1
x[1]	4	0	y[1]	52	1
x[2]	8	0	y[2]	56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0

在 x 结尾加了填充，x[i]和 y[i]现在就映射到了不同的组，消除了抖动冲突不命中。

练习题 6.7

在前面 dotprod 的例子中，在我们对数组 x 做了填充之后，所有对 x 和 y 的引用的命中率是多少？

旁注：为什么用中间的位来做索引？

你也许会奇怪，为什么高速缓存用中间的位来作为组索引，而不是用高位。为什么用中间的位更好，是有很好的原因的。图 6.31 说明了为什么。如果高位用做索引，那么一些连续的存储器块就会映射到相同的高速缓存块。例如，在图中，头四个块映射到第一个高速缓存组，第二个四个块映射到第二个组，依此类推。如果一个程序有良好的空间局部性，顺序扫描一个数组的元素，那么在任何时刻，高速缓存都只保存着一个块大小的数组内容。这样对高速缓存的使用效率很低。与以中间位作为索引相比，相邻的块总是映射到不同的高速缓存行。在这里的示例情况中，高速缓存能够存放整个 C 大小的数组内容，这里 C 是高速缓存的大小。

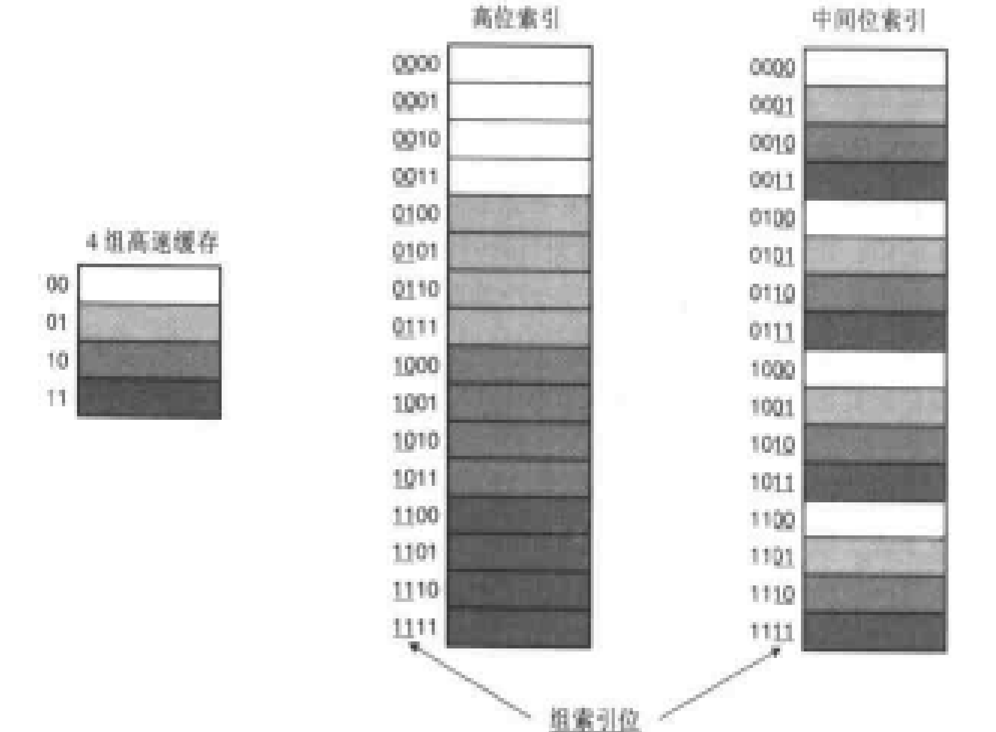


图 6.31　为什么用中间位来作为高速缓存的索引

练习题 6.8

一般而言，如果一个地址的高 s 位被用做组索引，那么连续的存储器块被映射到同一个高速缓存组。

A. 每个这样的连续的数组组块（chunks）中有多少个块？

B. 考虑下面的代码，它运行在一个高速缓存形式为（S,E,B,m）=（512,1,32,32）的系统上：

```
int    array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

在任意时刻，存储在高速缓存中的数组块的最大数量为多少？

6.4.3 组相联高速缓存

直接映射高速缓存中冲突不命中造成的问题是源于每个组只有一行（或者，按照我们的术语来描述就是 $E = 1$）这个限制。组相联高速缓存（set associative cache）放松了这条限制，所以每个组都保存有多于一个的高速缓存行。一个 $1 < E < C/B$ 的高速缓存通常被称为 E 路组相联高速缓存。在下一节中，我们会讨论 $E = C/B$ 这种特殊情况。图 6.23 展示了一个 2 路组相联高速缓存的结构。

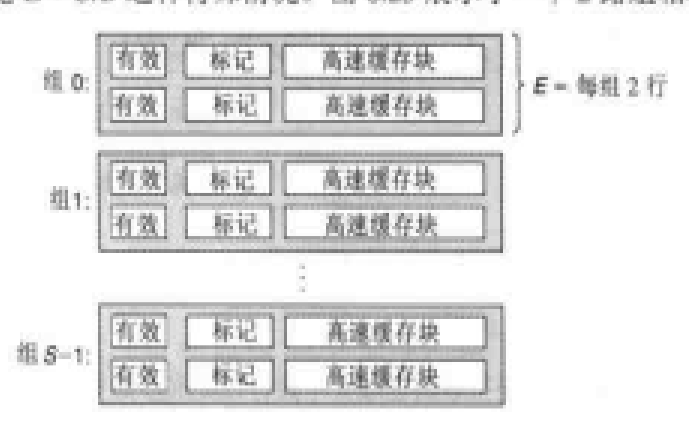


图 6.32 组相联高速缓存（$1 < E < C/B$）

在一个组相联高速缓存中，每个组包含多于一个的行。这里的这个示例是一个 2 路组相联高速缓存。

组相联高速缓存中的组选择

它的组选择与直接映射高速缓存的组选择一样，组索引位标识组。图 6.33 总结了这个原理。

组相联高速缓存中的行匹配和字选择

组相联高速缓存中的行匹配比直接映射高速缓存中的更复杂，因为它必须检查多个行的标记位和有效位，以确定所请求的字是否在集合中。一个传统的存储器是一个值的数组，以地址作为输入，并返回存储在那个地址的值。另一方面，一个相联的存储器是一个（key,value）对的数组，以 key 为输入，返回与输入的 key 相匹配的（key,value）对中的 value 值。因此，我们可以把组相联高速缓存中的每个组都看成一个小的组相联存储器，key 是标记和有效位，而 value 就是块的内容。

图 6.34 展示了组相联高速缓存中行匹配的基本思想。这里的一个重要思想就是组中的任何一行都可以包含任何映射到这个组的存储器块。所以高速缓存必须搜索组中的每一行，寻找一个有效的行，其标记与地址中的标记相匹配。如果高速缓存找到了这样一行，那么我们就命中，块偏移从这个块中选择一个字，和前面一样。

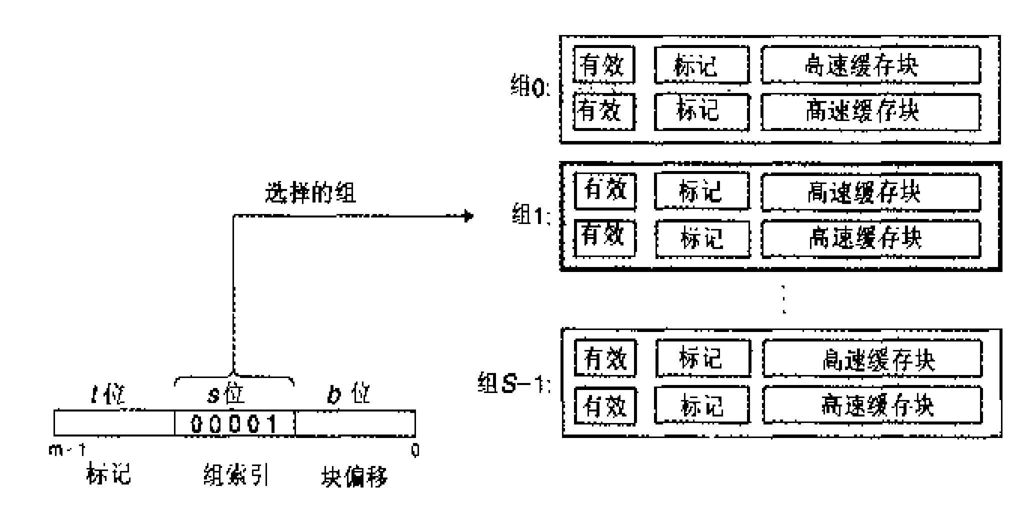


图 6.33 组相联高速缓存中的组选择

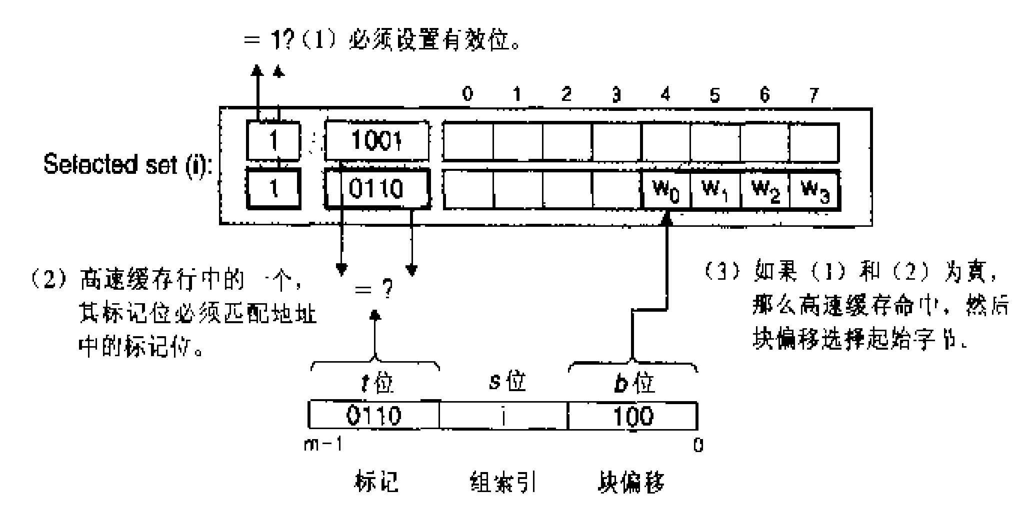


图 6.34 组相联高速缓存中的行匹配和字选择

组相联高速缓存中不命中时的行替换

如果CPU请求的字不在组的任何一行中，那么就是不命中，高速缓存必须从存储器中取出包含这个字的块。不过，一旦高速缓存取出了这个块，该替换哪个行呢？当然，如果有一个空行，那它就是个很好的候选。但是如果该组中没有空行，那么我们必须从中选择一个，希望CPU不会很快引用这个被替换的行。

程序员很难在他们代码中利用高速缓存替换策略，所以在此我们不会过多地讲述其细节。最简单的替换策略是随机选择要替换的行。其他更复杂的策略利用了局部性原理，以使在比较近的将来引用被替换的行的概率最小。例如，最不常使用（least-frequently-used，LFU）策略会替换在过去某个时间窗口内引用次数最少的那一行。最近最少使用（least-recently-used，LRU）策略会替换最后一次访问时间最久远的那一行。所有这些策略都需要额外的时间和硬件。但是，越往存储器层次结构下面走，远离CPU，一次不命中的开销就会更加昂贵，用更好的替换策略使得不命中最少也变得更加值得了。

6.4.4 全相联高速缓存

一个全相联高速缓存（fully associative cache）是由一个包含所有高速缓存行的组（也就是，$E = C/B$）组成的。图6.35给出了基本结构。

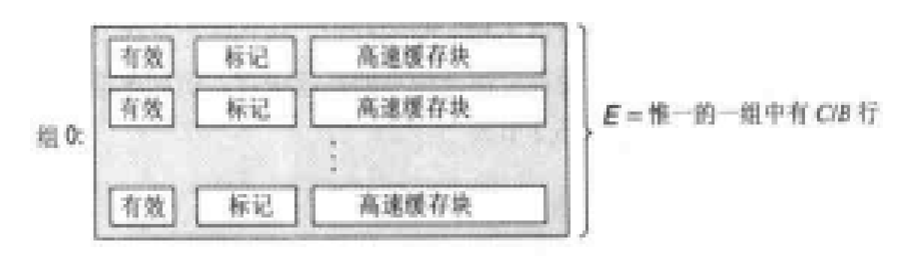


图 6.35 全组相联高速缓存（E = C / B）

在一个全相联高速缓存中，一个组包含所有的行。

全相联高速缓存中的组选择

全相联高速缓存中的组选择非常简单，因为只有一个组，图 6.36 做了个小结。注意地址中没有组索引位，地址只被划分成了一个标记和一个块偏移。

全相联高速缓存中的行匹配和字选择

全相联高速缓存中的行匹配和字选择与一个相联高速缓存中的是一样的，如图 6.37 所示。它们之间的区别主要是个规模大小的问题。

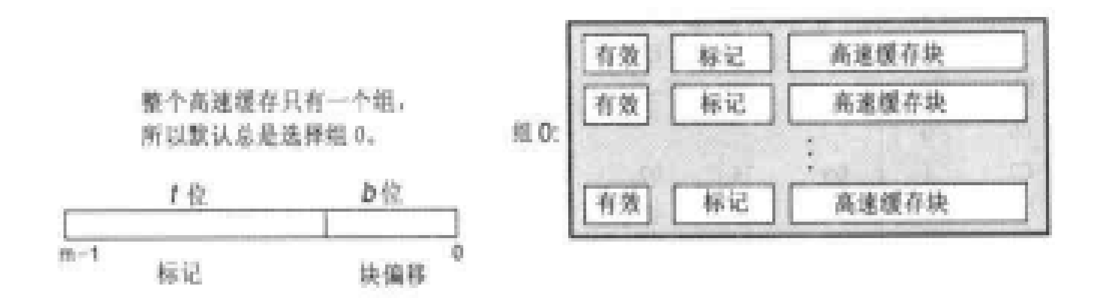


图 6.36 全相联高速缓存中的组选择

注意没有组索引位。

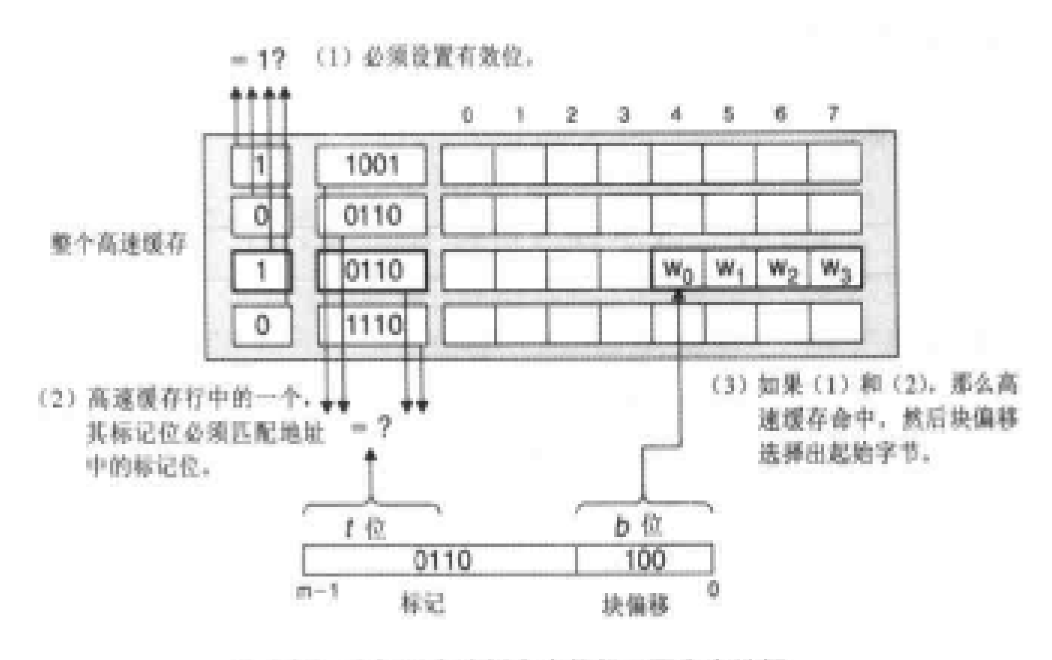


图 6.37 全相联高速缓存中的行匹配和字选择

因为高速缓存电路必须并行地搜索许多相匹配的标记，构造一个又大又快的相联高速缓存很困难，而且很昂贵。因此，全相联高速缓存只适合做小的高速缓存，例如虚拟存储器系统中的翻译备用缓冲器（TLB），它缓存页表项（10.6.2 节）。

练习题 6.9

下面的问题能帮助你加强理解高速缓存是如何工作的。有如下假设：

- 存储器是字节寻址的。
- 存储器访问的是 1 字节宽的字（不是 4 字节宽的字）。
- 地址的宽度为 13 位。
- 高速缓存是 2 路组相联的（$E=2$），块大小为 4 字节（$B=4$），有 8 个组（$S=8$）。

高速缓存的内容如下，所有的数字都是以十六进制来表示的：

2 路组相联高速缓存												
	行 0						行 1					
组索引	标记位	有效位	字节 0	字节 1	字节 2	字节 3	标记位	有效位	字节 0	字节 1	字节 2	字节 3
0	09	1	86	30	3F	10	00	0	—	—	—	—
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	—	—	—	—	0B	0	—	—	—	—
3	06	0	—	—	—	—	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	—	—	—	—
6	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

下面的方框展示的是地址格式（每个小方框一个位）。指出（在图中标出）用来确定下列内容的字段：

CO 高速缓存块偏移

CI 高速缓存组索引

CT 高速缓存标记

12	11	10	9	8	7	6	5	4	3	2	1	0

练习题 6.10

假设一个程序运行在图 6.9 中的机器上，它引用地址 0x0E34 处的 1 字节字。指出访问的高速缓存项目和十六进制表示的返回的高速缓存字节值。指出是否会发生缓存不命中。如果会出现缓存不命中，用“-”来表示“返回的高速缓存字节”。

A. 地址格式（每个小方框一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0

B. 存储器引用：

参数	值
高速缓存块偏移（CO）	0x
高速缓存组索引（CI）	0x
高速缓存标记（CT）	0x
高速缓存命中？（Y/N）	
返回的高速缓存字节	0x

练习题 6.11

对于存储器地址 0x0DD5，再做一遍练习题 6.10。

A. 地址格式（每个小方框一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0

B. 存储器引用：

参数	值
高速缓存块偏移（CO）	0x
高速缓存组索引（CI）	0x
高速缓存标记（CT）	0x
高速缓存命中？（Y/N）	
返回的高速缓存字节	0x

练习题 6.12

对于存储器地址 0x1FE4，再做一遍练习题 6.10。

A. 地址格式（每个小方框一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0

B. 存储器引用：

参数	值
高速缓存块偏移（CO）	0x
高速缓存组索引（CI）	0x
高速缓存标记（CT）	0x
高速缓存命中？（Y/N）	
返回的高速缓存字节	0x

练习题 6.13

对于练习题 6.9 中的高速缓存，列出所有的在组 3 中会命中的十六进制存储器地址。

6.4.5 有关写的问题

正如我们看到的，高速缓存关于读的操作非常简单。首先，在高速缓存中查找所需字 w 的拷贝。如果命中，立即返回字 w 给 CPU。如果不命中，从存储器中取出包含字 w 的块，将这个块存储到某个高速缓存行中（可能会驱逐一个有效的行），然后将字 w 返回给 CPU。

写的情况就要复杂一些了。假设 CPU 写一个已经缓存了的字 w［**写命中**（write hit）］。在高速缓存更新了它的 w 的拷贝之后，怎么更新 w 在存储器中的拷贝呢？最简单的方法，称为**直写**（write-through），就是立即将 w 的高速缓存块写回到存储器中。虽然简单，但是直写的缺点是每条存储指令都会引起总线上的一个写事务。另一种方法，称为**写回**（write-back），尽可能地推迟存储器更新，只有当替换算法要驱逐已更新的块时，才把它写到存储器。由于局部性，写回能显著地减少总线事务的数量，但是它的缺点是增加了复杂性。高速缓存必须为每个高速缓存行维护一个额外的**修改位**（dirty bit），表明这个高速缓存块是否被修改过。

另一个问题是如何处理写不命中。一种方法，称为**写分配**（write-allocate），加载相应的存储器块到高速缓存中，然后更新这个高速缓存块。写分配试图利用写的空间局部性，但是缺点是每次不命中都会导致一个块从存储器传送到高速缓存。另一种方法，称为**非写分配**（not-write-allocate），避开高速缓存，直接把这个字写到存储器中。直写高速缓存通常是非写分配的。写回高速缓存通常是写分配的。

为写操作优化高速缓存是一个细致而困难的问题，在此我们只略讲皮毛。细节随系统的不同而不同，而且通常是私有的，文档记录不详细。对于试图编写高速缓存比较友好的程序的程序员来说，我们建议在心里采用一个使用写回和写分配的高速缓存的模型，这样建议有几个原因。

通常，由于较长的传送时间，存储器层次结构中较低层的缓存更可能使用写回，而不是直写。例如，虚拟存储器系统（用主存作为存储在磁盘上的块的缓存）只使用写回。但是由于逻辑电路密度的提高，写回的高复杂性也越来越不成为阻碍了，我们在现代系统的所有层次上都能看到写回高速缓存。所以这种假设符合当前的趋势。假设使用写回写分配方法的另一个原因是，它与处理读的方式相对称，因为写回写分配试图利用局部性。因此，我们可以在高层次上开发我们的程序，展示良好的空间和时间局部性，而不是试图为某一个存储器系统进行优化。

6.4.6 指令高速缓存和统一的高速缓存

到目前为止，我们一直假设高速缓存只保存程序数据。不过，实际上，高速缓存既保存数据，也保存指令。只保存指令的高速缓存称为 *i-cache*。只保存程序数据的高速缓存称为 *d-cache*。既保存指令又包括数据的高速缓存称为**统一的高速缓存**（unified cache）。一个典型的桌面系统 CPU 芯片本身就包括一个 L1 i-cache 和一个 L1 d-cache。图 6.38 总结了基本的结构。

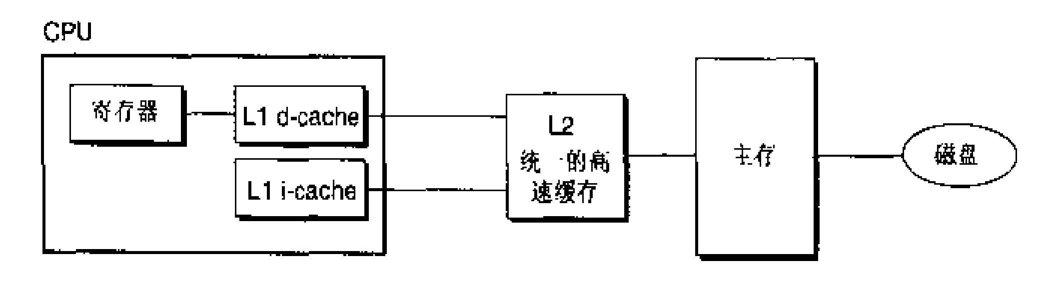


图 6.38 一个典型的多层高速缓存结构

有些高端系统，例如那些基于 Alpha 21164 的系统，将 L1 和 L2 高速缓存放在 CPU 芯片上，还有一个附加的、不在芯片上的 L3 高速缓存。为了提高性能，现代处理器包括分离的在芯片上的 i-cache 和 d-cache。有两个独立的高速缓存，处理器能够在同一个时钟周期中读一个指令字和一个数据字。就我们所知，没有系统使用了 L4 高速缓存，虽然处理器和存储器的速度差异在持续变大，也好像不太可能出现这样的情况。

旁注：真实系统有哪种高速缓存结构？

Intel Pentium 系统使用的高速缓存结构如图 6.38 所示，有一个在芯片上的 L1 i-cache，一个在芯片上的 L1 d-cache，还有一个不在芯片上的 L2 统一高速缓存。图 6.39 总结了这些高速缓存的基本参数。

高速缓存类型	相联度（E）	块大小（B）	组数（S）	高速缓存大小（C）
在芯片上的 L1 i-cache	4	32 B	128	16KB
在芯片上的 L1 d-cache	4	32 B	128	16KB
不在芯片上的 L2 统一高速缓存	4	32 B	1024～16384	128KB～2MB

图 6.39 Intel Pentium 高速缓存结构

6.4.7 高速缓存参数的性能影响

有许多指标来衡量高速缓存的性能：

- 不命中率（miss rate）。在一个程序执行或程序的一部分执行期间，存储器引用不命中的比率。它是这样计算的：不命中数量/引用数量。
- 命中率（hit rate）。命中的存储器引用比率。它等于 1——不命中率。
- 命中时间（hit time）。从高速缓存传送一个字到 CPU 所需的时间，包括组选择、行确认和字选择的时间。对于 L1 高速缓存来说，命中时间典型的是 1～2 个时钟周期。
- 不命中处罚（miss penalty）。由于不命中所需要的额外的时间。L1 不命中需要从 L2 得到服务的处罚，典型是 5～10 个周期。L1 不命中需要从主存得到服务的处罚，典型是 25～100 个周期。

优化高速缓存的成本和性能的折中是一项很精细的工作，它需要在现实的基准程序代码上进行大量的模拟，因此超出了我们讨论的范围。不过，还是可以认识一些定性的折中。

高速缓存大小的影响

一方面，较大的高速缓存可能会提高命中率；另一方面，使得大存储器运行得更快总是要难一些的。结果，较大的高速缓存可能会增加命中时间。对于芯片上的 L1 高速缓存来说这一点尤为重要，因为它的命中时间必须为一个时钟周期。

块大小的影响

大的块有利有弊。一方面，较大的块能利用程序中可能存在的空间局部性，帮助提高命中率。不过，对于给定的高速缓存大小，块越大就意味着高速缓存行数越少，这会损害时间局部性比空间局部性更好的程序中的命中率。较大的块对不命中处罚也有负面影响，因为块越大，传送时间就越长。现代系统通常会折中，使高速缓存块包含 4～8 个字。

相联度的影响

这里的问题是参数E（每个组中高速缓存行数）的选择的影响。较高的相联度（也就是E的值较大）的优点是降低了高速缓存由于冲突不命中出现抖动的可能性。不过，较高的相联度会造成较高的成本。较高的相联度实现起来很昂贵，而且很难使之速度变快。每一行需要更多的标记位，每一行需要额外的LRU状态位和额外的控制逻辑。较高的相联度会增加命中时间，因为复杂性增加了。另外，还会增加不命中处罚，因为选择牺牲行（victim line）的复杂性也增加了。

相联度的选择最终变成了命中时间和不命中处罚之间的折中。传统上，努力争取时钟频率的高性能系统会选择直接映射L1高速缓存（这里的不命中处罚只是几个周期），而在较低层上使用比较小的相联度（比如说2~4），但是没有固定的规则。Intel Pentium 系统中，L1 和 L2 高速缓存是4路组相联的。Alpha 21164 系统中，L1 指令和数据高速缓存是直接映射的，L2 高速缓存是3路组相联的，而L3 高速缓存是直接映射的。

写策略的影响

直写高速缓存比较容易实现，而且能使用写缓冲区（write buffer），它独立于高速缓存，用来更新存储器。此外，读不命中开销没这么大，因为它们不会触发存储器写。另一方面，写回高速缓存引起的传送比较少，它允许更多的到存储器的带宽用于执行DMA的I/O设备。此外，越往层次结构下面走，传送时间增加，减少传送的数量就变得更加重要。一般而言，高速缓存越往下层，越可能使用写回而不是直写。

旁注：高速缓存行、组和块有什么区别？

很容易混淆高速缓存行、组和块之间的区别。让我们来回顾一下这些概念，确保概念清晰：

- 块是一个固定大小的信息包，在高速缓存和主存（或下一层高速缓存）之间来回传送。
- 行是高速缓存中存储块以及其他信息（例如有效位和标记位）的容器。
- 组是一个或多个行的集合。直接映射高速缓存中的组只由一行组成。组相联和全相联高速缓存中的组是由多个行组成的。

在直接映射高速缓存中，组和行确实是等价的。不过，在相联高速缓存中，组和行是很不一样的，这两个词不能互换使用。

因为一行总是存储一个块，术语“行”和“块”通常互换使用。例如，系统专家总是说高速缓存的“行大小”，实际上他们指得是块大小。这样的用法十分普遍，只要你理解块和行之间的区别，它不会造成任何误会。

6.5 编写高速缓存友好的代码

在6.2节中，我们介绍了局部性的思想，而且大概地谈了一下什么会具有良好的局部性。既然我们已经明白了高速缓存存储器是如何工作的了，我们就能更加精确一些了。局部性比较好的程序更容易有较低的不命中率，而不命中率较低的程序倾向于比不命中率较高的程序运行得更快。因此，从具有良好局部性的意义上来说，好的程序员总是应该试着去编写*高速缓存友好*（cache friendly）的代码。下面就是我们用来确保我们的代码高速缓存友好的基本方法：

1. 让最常见的情况运行得快。程序通常把大部分时间都花在少量的核心函数上，而这些函数通

常把大部分时间都花在了少量循环上。所以要把注意力集中在核心函数里的循环上，而忽略其他部分。

2. *在每个循环内部使缓存不命中数量最小。*在其他条件，例如加载和存储的总次数，相同的情况下，不命中率较低的程序运行得更快。

为了看看实际上这是怎么工作的，考虑 6.2 节中的函数 sumvec：

```
1   int sumvec(int v[N])
2   {
3       int i, sum = 0;
4
5       for (i = 0; i < N; i++)
6           sum += v[i];
7       return sum;
8   }
```

这个函数高速缓存友好吗？首先，注意对于局部变量 i 和 sum，循环体有良好的时间局部性。实际上，因为它们都是局部变量，任何合理的优化编译器都会把它们缓存在寄存器文件中，也就是存储器层次结构的最高层中。现在考虑一下对向量 v 的步长为 1 的引用。一般而言，如果一个高速缓存的块大小为 B 字节，那么一个步长为 k 的引用模式（这里 k 是以字为单位的）平均每次循环迭代会有 min(1, (wordsize×k)/B)次缓存不命中。当 k=1 时，它取最小值，所以对 v 的步长为 1 的引用确实是高速缓存友好的。例如，假设 v 是块对齐的，字为 4 个字节，高速缓存块为 4 个字，而高速缓存初始为空（冷高速缓存）。然后，无论是什么样的高速缓存结构，对 v 的引用都会得到下面的命中和不命中模式：

v[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
访问顺序，命中[h]或不命中[m]	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]

在这个例子中，对 v[0]的引用会不命中，而相应的包含 v[0]~v[3]的块会被从存储器加载到高速缓存中。因此，接下来三个引用都会命中。对 v[4]的引用会导致不命中，而一个新的块被加载到高速缓存中，接下来的三个引用都命中，依此类推。一般而言，四个引用中的三个会命中，在这种冷缓存的情况下，这是我们所能做到的最好的情况了。

总之，我们简单的 sumvec 示例说明了两个关于编写高速缓存友好的代码的重要问题：

- 对局部变量的反复引用是好的，因为编译器能够将它们缓存在寄存器文件中（时间局部性）。
- 步长为 1 的引用模式是好的，因为存储器层次结构中所有层次上的缓存都是将数据存储为连续的块（空间局部性）。

在对多维数组进行操作的程序中，空间局部性尤其重要。例如，考虑 6.2 节中的 sumarrayrows 函数，它按照行优先顺序对一个二维数组的元素求和：

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
```

```
6            for (j = 0; j < N; j++)
7                sum += a[i][j];
8        return sum;
9    }
```

由于 C 以行优先顺序存储数组，所以这个函数中的循环有与 sumvec 一样好的步长为 1 的访问模式。例如，假设我们对这个高速缓存做与对 sumvec 一样的假设。那么对数组 a 的引用会得到下面的命中和不命中模式：

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i=1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i=2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i=3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

但是如果我们做一个看似无伤大雅的改变——交换循环的次序，看看会发生什么：

```
1    int sumarraycols(int a[M][N])
2    {
3        int i, j, sum = 0;
4
5        for (j = 0; j < N; j++)
6            for (i = 0; i < M; i++)
7                sum += a[i][j];
8        return sum;
9    }
```

在这种情况中，我们是一列一列而不是一行一行地扫描数组的。如果我们够幸运，整个数组都在高速缓存中，那么我们也会有相同的不命中率 1/4。不过，如果数组比高速缓存要大（更可能出现这种情况），那么每次对 a[i][j]的访问都会不命中！

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1 [m]	5 [m]	9 [m]	13 [m]	17 [m]	21 [m]	25 [m]	29 [m]
i=1	2 [m]	6 [m]	10 [m]	14 [m]	18 [m]	22 [m]	26 [m]	30 [m]
i=2	3 [m]	7 [m]	11 [m]	15 [m]	19 [m]	23 [m]	27 [m]	31 [m]
i=3	4 [m]	8 [m]	12 [m]	16 [m]	20 [m]	24 [m]	28 [m]	32 [m]

较高的命中率对运行时间可以有显著的影响。例如，在我们的桌面机器上，sumarraycols 每次迭代需要运行大约 20 个时钟周期，而 sumarrayrows 每次迭代需要运行大约 10 个周期。总之，程序员应该注意他们程序中的局部性，试着编写利用局部性的程序。

练习题 6.14

在信号处理和科学计算的应用中，转置矩阵的行和列是一个很重要的问题。从局部性的角度来看，它也很有趣，因为它的引用模式既是以行为主（row-wise）的，也是以列为主（column-wise）

的。例如，考虑下面的转置函数：

```
1    typedef int array[2][2];
2
3    void transpose1(array dst, array src)
4    {
5        int i, j;
6
7        for (i = 0; i < 2; i++) {
8            for (j = 0; j < 2; j++) {
9                dst[j][i] = src[i][j];
10           }
11       }
12   }
```

假设在一台具有如下属性的机器上运行这段代码：

- sizeof(int) = 4。
- src 数组从地址 0 开始，dst 数组从地址 16 开始（十进制）。
- 只有一个 L1 数据高速缓存，它是直接映射的、直写、写分配，块大小为 8 个字节。
- 这个高速缓存总的大小为 16 个数据字节，一开始是空的。
- 对 src 和 dst 数组的访问分别是读和写不命中的惟一原因。

A. 对每个 row 和 col，指明对 src[row][col]和 dst[row][col]的访问是命中（h）还是不命中（m）。例如，读 src[0][0]会不命中，写 dst[0][0]也不命中。

dst 数组		
	列 0	列 1
0 行	m	
1 行		

src 数组		
	列 0	列 1
0 行	m	
1 行		

B. 对于一个大小为 32 数据字节的高速缓存重复这个练习题。

练习题 6.15

最近一个很成功的游戏 SimAquarium 的核心就是一个紧密循环（tight loop），它计算 256 个海藻（algae）的平均位置。在一台具有块大小为 16 字节（B=16）、整个大小为 1024 字节的直接映射数据缓存的机器上测量它的高速缓存性能。定义如下：

```
1    struct algae_position {
2        int x;
3        int y;
4    };
5
6    struct algae_position grid[16][16];
7    int total_x = 0, total_y = 0;
8    int i, j;
```

还有如下假设：

- sizeof(int) = 4。
- grid 从存储器地址 0 开始。
- 这个高速缓存开始时是空的。
- 惟一的存储器访问是对数组 grid 的元素的访问。变量 i、j、total_x 和 total_y 存放在寄存器中。

确定下面代码的高速缓存性能:

```
1   for (i = 0; i < 16; i++) {
2       for (j = 0; j < 16; j++) {
3           total_x += grid[i][j].x;
4       }
5   }
6
7   for (i = 0; i < 16; i++) {
8       for (j = 0; j < 16; j++) {
9           total_y += grid[i][j].y;
10      }
11  }
```

A. 读总数是多少? ____________

B. 高速缓存不命中的读总数是多少? ____________

C. 不命中率是多少? ____________

练习题 6.16

给定练习题 6.15 的假设，确定下列代码的高速缓存性能:

```
1   for (i = 0; i < 16; i++){
2       for (j = 0; j < 16; j++) {
3           total_x += grid[j][i].x;
4           total_y += grid[j][i].y;
5       }
6   }
```

A. 读总数是多少? ____________

B. 高速缓存不命中的读总数是多少? ____________

C. 不命中率是多少? ____________

D. 如果高速缓存有两倍大，那么不命中率会是多少呢? ____________

练习题 6.17

给定练习题 6.15 的假设，确定下列代码的高速缓存性能:

```
1   for (i = 0; i < 16; i++){
2       for (j = 0; j < 16; j++) {
3           total_x += grid[i][j].x;
4           total_y += grid[i][j].y;
5       }
```

```
6   }
```

A. 读总数是多少？ ______

B. 高速缓存不命中的读总数是多少？ ______

C. 不命中率是多少？ ______

D. 如果高速缓存有两倍大，那么不命中率会是多少呢？ ______

6.6 综合：高速缓存对程序性能的影响

本节通过研究高速缓存对运行在实际机器上的程序的性能影响，综合了我们对存储器层次结构的讨论。

6.6.1 存储器山（memory mountain）

一个程序从存储系统中读数据的速率被称为**读吞吐率**（read throughput），或者有时称为**读带宽**（read bandwidth）。如果一个程序在 s 秒的时间段内读 n 个字节，那么这段时间内的读吞吐率就等于 n/s，典型地是以兆字节每秒（MB/s）为单位的。

如果我们要编写一个程序，它从一个紧密程序循环（tight program loop）中发出一系列读请求，那么测量出的读吞吐率能让我们看到对于这个读序列来说的存储系统的性能。图 6.40 给出了一对测量某个读序列吞吐率的函数。

code/mem/mountain/mountain.c

```
1   void test(int elems, int stride) /* The test function */
2   {
3       int i, result = 0;
4       volatile int sink;
5
6       for (i = 0; i < elems; i += stride)
7           result += data[i];
8       sink = result; /* So compiler doesn't optimize away the loop */
9   }
10
11  /* Run test(elems, stride) and return read throughput (MB/s) */
12  double run(int size, int stride, double Mhz)
13  {
14      double cycles;
15      int elems = size / sizeof(int);
16
17      test(elems, stride);                        /* warm up the cache */
18      cycles = fcyc2(test, elems, stride, 0);     /* call test(elems,stride) */
19      return (size / stride) / (cycles / Mhz);    /* convert cycles to MB/s */
20  }
```

code/mem/mountain/mountain.c

图 6.40 测量和计算读吞吐率的函数

test 函数通过以步长 stride 扫描整数数组的头 elems 个元素来产生读序列。run 函数是一个包装函数（wrapper），它调用 test 函数，并返回测量出的读吞吐率。第 18 行的 fcyc2 函数（没有显示出来）估计了 test 函数的运行时间，以 CPU 周期为单位，使用的是第 9 章中讲述的 K 次最优（K-best）测量方法。注意，run 函数的参数 size 是以字节为单位的，而 test 函数对应的参数 elems 是以字为单位的。另外，注意第 19 行将 MB/s 计算为 10^6 字节/秒，而不是 2^{20} 字节/秒。

run 函数的参数 size 和 stride 允许我们控制产生出的读序列的局部性程度。size 的值越小，得到的工作集越小，因此时间局部性越好。stride 的值越小，得到的空间局部性越好。如果我们反复以不同的 size 和 stride 值调用 run 函数，那么我们就能覆盖读带宽的一个时间和空间局部性的二维函数，称为存储器山（memory mountain）。图 6.41 展示了一个称为 mountain 的程序，它生成存储器山。

code/mem/mountain/mountain.c

```
1   #include <stdio.h>
2   #include "fcyc2.h" /* K-best measurement timing routines */
3   #include "clock.h" /* routines to access the cycle counter */
4
5   #define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
6   #define MAXBYTES (1 << 23) /* ... up to 8 MB */
7   #define MAXSTRIDE 16       /* Strides range from 1 to 16 */
8   #define MAXELEMS MAXBYTES  /sizeof(int)
9
10  int data[MAXELEMS];        /* The array we'll be traversing */
11
12  int main()
13  {
14      int size;          /* Working set size (in bytes) */
15      int stride;        /* Stride (in array elements) */
16      double Mhz;        /* Clock frequency */
17
18      init_data(data, MAXELEMS);    /* Initialize each element in data to 1 */
19      Mhz = mhz(0);                 /* Estimate the clock frequency */
20      for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
21          for (stride = 1; stride <= MAXSTRIDE; stride++) {
22              printf("%.1f\t", run(size, stride, Mhz));
23          }
24          printf("\n");
25      }
26      exit(0);
27  }
```

code/mem/mountain/mountain.c

图 6.41 mountain：一个生成存储器山的程序

mountain 程序以不同的工作集大小和步长调用 run 函数。工作集大小从 1KB 开始，每次增加一

倍，最大值到8MB。步长范围为1～16。对于每个工作集大小和步长的组合，mountain打印出读吞吐率，单位为MB/s。第19行的mhz函数（没有显示出来）是一个依赖于系统的函数，它使用第9章中讲述的技术，估计CPU时钟频率。

每台计算机都有惟一的存储器山，存储器山刻画了计算机的存储系统的能力。例如，图6.42展示了一个Intel Pentium III Xeon系统的存储器山。

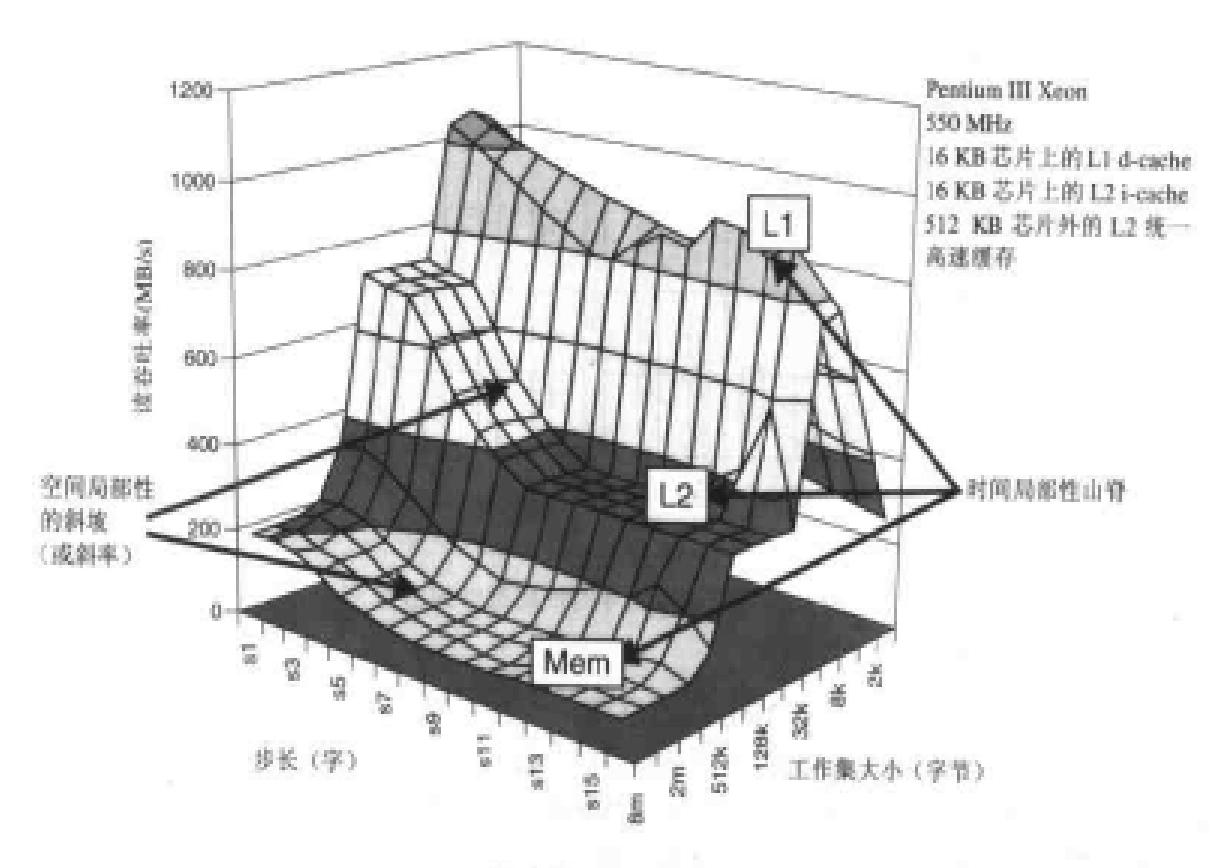


图 6.42 存储器山

这座Xeon山的地形地势展现了一个很丰富的结构。垂直于工作集大小轴的是三条山脊，分别对应于工作集完全在L1高速缓存、L2高速缓存和主存内的时间局部性区域。注意，L1山脊的最高点（那里CPU读速率为1GB/s）与主存山脊的最低点（那里CPU读速率为80MB/s）之间的差别有一个数量级。

L1山脊有两点特性应该指出来。首先，对于一个常数步长，注意读吞吐率如何随着工作集大小从16KB降到1KB骤然下降的（在山脊的背面下降的）。其次，对于工作集大小16KB，L1山脊线的峰顶随着步长的增加而降低。因为L1高速缓存保存着整个工作集，所以这些特性不能反映L1高速缓存的真实性能。它们是调用test函数和准备执行循环的开销的结果。沿着L1山脊，对于小的工作集来说，这些开销没有像使用较大工作集时那样得到补偿。

在L2和主存山脊上，随着步长的增加，有一个空间局部性的斜坡，沿着山下降。L2上的斜坡最陡，这是因为当L2高速缓存不得不从主存传送块时遭受的绝对不命中处罚很大。注意，即使是当工作集太大，不能全部装进L1或L2高速缓存时，主存山脊的最高点也比它的最低点高两倍。因此，即使是当程序的时间局部性很差时，空间局部性仍然能补救，并且是非常重要的。

如果我们从这座山中取出一个片段，保持步长为常数，如图6.43，我们就能很清楚地看到高速缓存的大小和时间局部性对性能的影响了。对于大小最大为16KB并包括16KB，工作集完全能放进L1 d-cache中，因此，在吞吐率峰值1 GB/s处，读都是由L1来服务的。

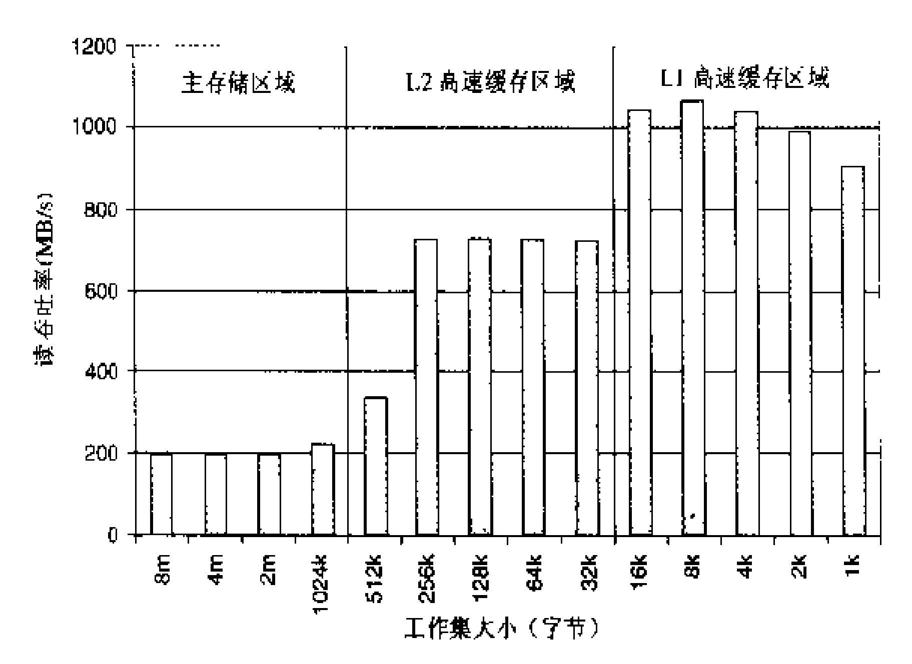


图 6.43 存储器山中时间局部性的山脊

这幅图展示了图6.42中stride=1时的一个片段。

对于大小最大为256KB并包括256KB，工作集完全能放进L2统一高速缓存中。再大的工作集主要就由主存来服务了。256KB～512KB之间读吞吐率的下降很有趣。因为L2高速缓存是512KB，我们可能会预测下降出现在512KB处，而不是256KB处。要确认的惟一方法就是进行一次详细的高速缓存模拟，不过我们怀疑原因在于Pentium III的L2高速缓存是一个统一的高速缓存，既保存指令又保存数据。我们可能会看到的是L2中指令和数据之间的冲突不命中的结果，它使得不能将整个数组都放到L2高速缓存中。

以相反的方向横切这座山，保持工作集大小不变，我们从中能看到空间局部性对读吞吐率的影响。例如，图6.44展示了工作集大小固定为256KB时的片段。这个片段是沿着图6.42中的L2山脊切的，这里，工作集完全能够放到L2高速缓存中，但是对L1高速缓存来说太大了。

注意随着步长从1个字增长到8个字，读吞吐率是如何平稳地下降的。在山的这个区域中，L1中的读不命中会导致一个块从L2传送到L1。取决于步长，后面在L1中这个块上会有一定数量的命中。随着步长的增加，L1不命中与L1命中的比值也增加了。因为不命中服务起来要比命中慢一些，所以读吞吐率也下降了。一旦步长达到了8个字，在这个系统上8个字就等于块的大小了，每个读请求在L1中都会不命中，必须从L2服务。因此，对于步长至少为8个字的读吞吐率是一个常数速率，是由从L2传送高速缓存块到L1的速率决定的。

总结一下我们对存储器山的讨论，存储器系统的性能不是一个数字就能描述的。相反，它是一座时间和空间局部性的山，这座山的上升高度差别可以超过一个数量级。明智的程序员会试图构造他们的程序，使得程序运行在山峰而不是低谷。目标就是利用时间局部性，使得频繁使用的字从L1

中取出，还要利用空间局部性，使得尽可能多的字从一个 L1 高速缓存行中访问到。

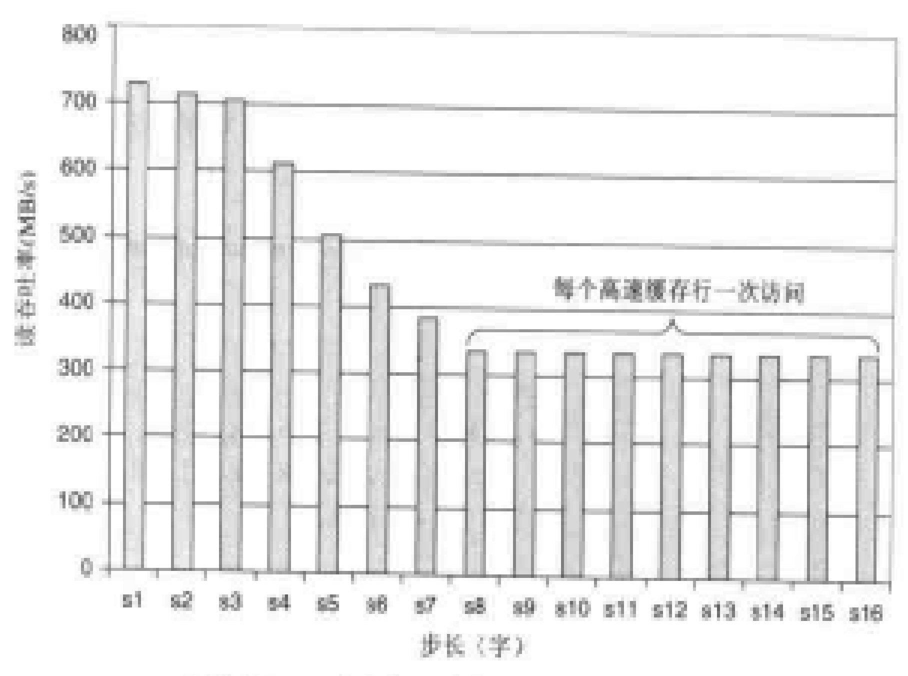


图 6.44　一个空间局部性的斜坡（或斜率）

这幅图展示了图 6.42 中 size=256 KB 时的一个片段。

练习题 6.18

图 6.42 中的存储器山有两个轴：步长和工作集大小。哪个轴对应于空间局部性？哪个轴对应于时间局部性？

练习题 6.19

作为关心性能的程序员，知道对存储器层次结构各个部分访问时间的粗略估计值是很重要的。使用图 6.42 中的存储器山，估计从下列这些位置读出一个 4 字节的字所需的时间，以 CPU 周期为单位：

A. 在芯片上的 L1 d-cache。

B. 不在芯片上的 L2 高速缓存。

C. 主存。

假设在（工作集大小=16M，步长=16）时，读吞吐率为 80MB/s。

6.6.2　重新排列循环以提高空间局部性

考虑一对 $n \times n$ 矩阵相乘的问题：$C = AB$。例如，如果 $n = 2$，那么

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

这里

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \end{aligned}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

矩阵乘法函数通常是用三个嵌套的循环来实现的，分别用索引分别是 i、j 和 k 来标识。如果我们改变循环的次序，对代码进行一些其他的小改动，我们就能得到矩阵乘法的六个在功能上等价的版本，如图 6.45 所示。每个版本都以它循环的顺序来惟一地标识。

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

（a）*ijk* 版本

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (i = 0; i < n; i++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

（b）*jik* 版本

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

（c）*jki* 版本

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (j = 0; j < n; j++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
```

```
5                C[i][j] += A[i][k]*r;
6        }
```

code/mem/matmult/mm.c

(d) *kji* 版本

code/mem/matmult/mm.c

```
1    for (k = 0; k < n; k++)
2        for (i = 0; i < n; i++) {
3            r = A[i][k];
4            for (j = 0; j < n; j++)
5                C[i][j] += r*B[k][j];
6        }
```

code/mem/matmult/mm.c

(e) *kij* 版本

code/mem/matmult/mm.c

```
1    for (i = 0; i < n; i++)
2        for (k = 0; k < n; k++) {
3            r = A[i][k];
4            for (j = 0; j < n; j++)
5                C[i][j] += r*B[k][j];
6        }
```

code/mem/matmult/mm.c

(f) *ikj* 版本

图 6.45 矩阵乘法的六个版本

在高层来看，这六个版本是非常相似的。如果加法是可结合的，那么每个版本计算出的结果完全一样[2]。每个版本总共都执行 $O(n^3)$个操作，而加法和乘法的数量相同。A 和 B 的 n^2 个元素中的每一个都要读 n 次。计算 C 的 n^2 个元素中的每一个都要对 n 个值求和。不过，如果我们分析最里层循环迭代的行为，我们发现在访问数量和局部性上还是有区别的。为了这次分析的目的，我们做了如下假设：

- 每个数组都是一个 double 类型的 $n \times n$ 的数组，sizeof(double) = 8。
- 只有一个高速缓存，其块大小为 32 字节（B = 32）。
- 数组大小 n 很大，以至于矩阵的一行都不能完全装进 L1 高速缓存中。
- 编译器将局部变量存储到寄存器中，因此循环内对局部变量的引用不需要任何加载或存储指令。

图 6.46 总结了我们对循环的分析结果。注意六个版本成对地形成了三个等价类，用最内层循环中访问的矩阵对来表示每个类。例如，版本 ijk 和 jik 是类 AB 的成员，因为它们在最内层的循环中引用的是矩阵 A 和 B（而不是 C）。对于每个类，我们统计了每个内层循环迭代中加载（读）和存

2 正如我们在第 2 章中学到的，浮点加法是可交换的，但是通常不是可结合的。实际上，如果矩阵不把极大的数和极小的数混在一起——存储物理属性的矩阵常常这样，那么假设浮点加法是可结合的也是合理的。

储（写）的数量，每次循环迭代中对A、B和C的引用在高速缓存中不命中的数量，以及每次迭代高速缓存不命中的总数。

类AB例程的里层循环[图6.45（a）和（b）]以步长1扫描数组A的一行。因为每个高速缓存块保存四个双字，A的不命中率是每次迭代不命中0.25次。另一方面，里层循环以步长n扫描数组B的一列。因为n很大，每次对数组B的访问都会不命中，所以每次迭代总共会有1.25次不命中。

矩阵乘法版本（类）	加载每次迭代使用的	存储每次迭代使用的	A不命中每次迭代使用的	B不命中每次迭代使用的	C不命中每次迭代使用的	总不命中每次迭代使用的
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

图6.46 矩阵乘法里层循环的分析

六个版本分为三个等价类，以里层循环中访问的数组对来表示。

类AC例程的里层循环[图6.45（c）和（d）]有一些问题。每次迭代执行两个加载和一个存储（相对于类AB例程，它们执行2个加载而没有存储）。第二，里层循环以步长n扫描A和C的列。结果是每次加载都会不命中，所以每次迭代总共有两个不命中。注意，与类AB例程相比，交换循环降低了空间局部性。

BC例程[图6.45（e）和（f）]展示了一个很有趣的折中：使用了两个加载和一个存储，它们比AB例程多需要一个存储器操作。另一方面，因为里层循环以步长1访问模式扫描B和C的列，每次迭代每个数组上的不命中率只有0.25次不命中，所以每次迭代总共有0.50个不命中。

图6.47小结了一个Pentium III Xeon系统上矩阵乘法各个版本的性能。这个图画出了每次里层循环迭代所需的测量出的CPU周期数作为数组大小（n）的函数。

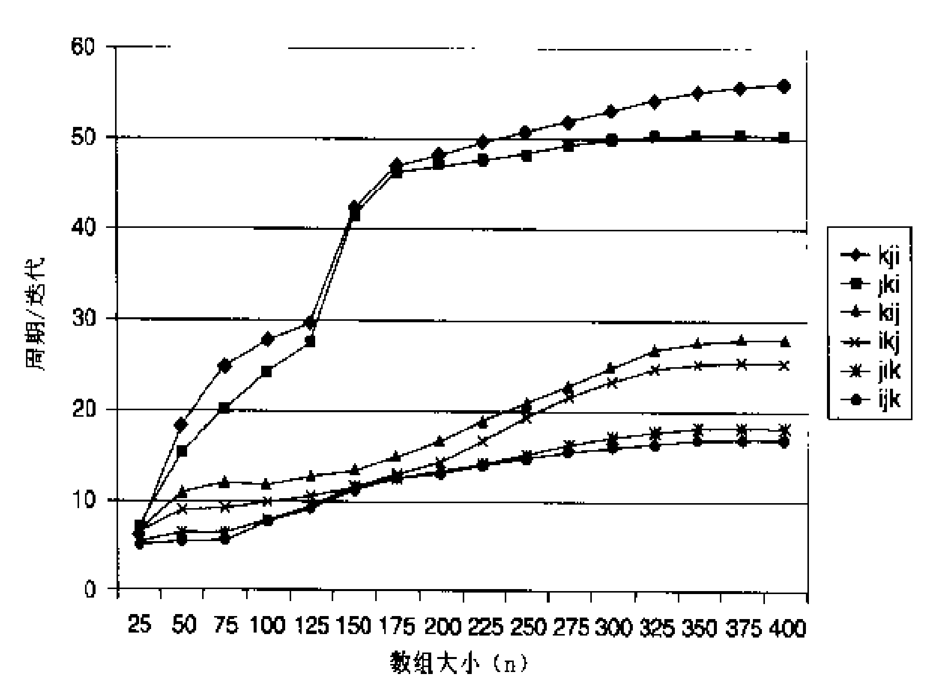


图6.47 Pentium III Xeon 矩阵乘法性能

图例：kji和jki：类AC；kij和ikj：类BC；ijk和jik：类AB。

对于这幅图有很多有意思的地方值得注意：

- 对于大的 n 值，即使每个版本都执行相同数量的浮点算术操作，最快的版本比最慢的版本运行得快三倍。
- 存储器访问数量和局部性都相同的版本，有大致相同的测量性能。
- 存储性能最糟糕的两个版本，就每次迭代的访问数量和不命中数量而言，明显地比其他四个版本运行得慢，其他四个版本有较少的不命中次数或者较少的访问次数，或者兼而有之。
- 类 AB 例程——每次迭代有 2 个存储器访问和 1.25 次不命中——在这种机器上运行得比类 BC 例程——每次迭代 3 个存储器访问和 0.5 次不命中——要好一点，后者用一个额外的存储器访问来换取较低的不命中率。要点就是对于性能来说，高速缓存不命中率并不是问题的全部。存储器访问的数量也很重要，而且在许多情况中，找到最好的性能就是要在这两者之间做出权衡。练习题 6.32 和 6.33 更深入地论述了这个问题。

6.6.3 使用分块来提高时间局部性

在上一节中，我们看到一些很简单的循环重新排列是如何能够提高空间局部性的。但是我们也看到，即使使用很好的循环嵌套，每次循环迭代的时间都随着数组大小的增长而增长。发生的事情是这样的，当数组大小增加时，时间局部性降低了，而高速缓存中容量不命中的数目增加了。为了改正这个问题，我们使用了一种普通的称为分块（blocking）的技术。不过我们必须指出，与那些为了提高空间局部性的简单循环变换不同，分块使得代码更难阅读和理解。因此，它最适合于优化编译器或者频繁执行的库例程。不过学习和理解这项技术仍然是很有趣的，因为它是一个能够产生巨大性能收益的很普通的概念。

分块的大致思想是将一个程序中的数据结构组织成称为块（block）的组块（chunks）。（在这个上下文中，“块”指得是一个应用级的数据组块，而不是高速缓存块。）这样构造程序，使得能够将一个块加载到 L1 高速缓存中，并在这个块中进行所需的所有的读和写，然后丢掉这个块，加载下一个块，依此类推。

分块一个矩阵乘法函数是这样进行的，将矩阵划分成子矩阵，然后利用可以像标量一样处理子矩阵这个数学依据。例如，如果 $n=8$，那么我们可以将每个矩阵划分成四个 4×4 的子矩阵：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

这里

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

图 6.48 展示了矩阵乘法的一个分块版本，我们称之为 bijk 版本。这段代码背后的基本思想是将 A 和 C 划分成 $1\times bsize$ 的行条（row slivers），将 B 划分成 $bsize\times bsize$ 的块。最内层的（j，k）循环对用 B 的一个块去乘以 A 的一个行条，将结果放到 C 的一个行条中。用 B 中同一个块，i 循环迭代通过 A 和 C 的 n 个行条。

code/mem/matmult/bmm.c

```
1   void bijk(array A, array B, array C, int n, int bsize)
2   {
3       int i, j, k, kk, jj;
4       double sum;
5       int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */
6
7       for (i = 0; i < n; i++)
8           for (j = 0; j < n; j++)
9               C[i][j] = 0.0;
10
11      for (kk = 0; kk < en; kk += bsize) {
12          for (jj = 0; jj < en; jj += bsize) {
13              for (i = 0; i < n; i++) {
14                  for (j = jj; j < jj + bsize; j++) {
15                      sum = C[i][j];
16                      for (k = kk; k < kk + bsize; k++) {
17                          sum += A[i][k]*B[k][j];
18                      }
19                      C[i][j] = sum;
20                  }
21              }
22          }
23      }
24  }
```

code/mem/matmult/bmm.c

图 6.48 分块的矩阵乘法

这个简单的版本假设数组大小（n）是块大小（bsize）的整数倍。

图 6.49 给出了图 6.48 中分块代码的一个图形化的说明。关键思想是它加载 B 的一个块到高速缓存中，使用它，然后丢弃它。对 A 的引用有很好的空间局部性，因为是以步长 1 来访问每个行条的。它也有很好的空间局部性，因为是连续 bsize 次引用整个行条的。对 B 的引用有好的时间局部性，因为是连续 n 次访问整个 bsize × bsize 块的。最后，对 C 的引用有好的空间局部性，因为行条的每个元素是连续写的。注意对 C 的引用没有好的时间局部性，因为每个行条都只被访问一次。

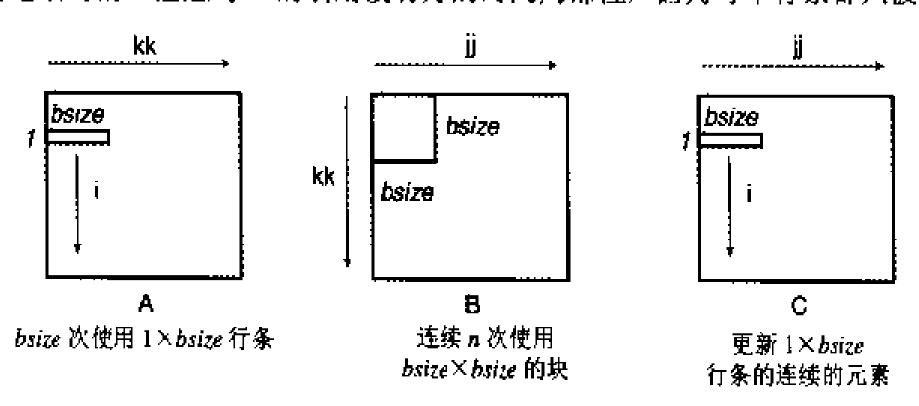


图 6.49 分块的矩阵乘法的图形化说明

最内层的 (j,k) 循环对用 B 的一个 bsize×bsize 的块去乘以 A 的一个 1×bsize 的行条，将结果放到 C 的一个 1×bsize 的行条中。

分块可能使代码更难阅读，但是它也能带来巨大的性能收益。图 6.50 展示了 Pentium III Xeon 系统上两个分块的矩阵乘法版本的性能（bsize = 25）。注意，分块使得运行时间比最好的非分块版本提高了 1 倍，从每次迭代大约 20 个周期改进到每次迭代大约 10 个周期。另一件关于分块的有趣事情是随着数组大小的增长，每次迭代的时间几乎保持不变。对于小的数组大小，分块版本中的额外开销使得它比非分块版本运行得还要慢。在大约 n=100 处有一个交叉点，在此之后分块版本就运行得更快了。

旁注：高速缓存和流媒体工作负载

实时处理网络视频和音频数据的应用变得越来越重要。在这些应用中，数据以来自某个输入设备（例如麦克风、照相机或者网络连接——参加第 12 章）的稳定的流的方式到达机器。当数据到达时，处理它们，然后发送到一个输出设备，并且最终丢弃掉，从而为新到达的数据腾出位置。

存储器层次结构有多适合这些流媒体（streaming media）工作负载呢？因为数据是在它们到达时顺序处理的，我们能够从空间局部性上获得些好处，就像我们 6.6 节中矩阵乘法的例子那样。不过，因为数据只被处理一次，然后就丢弃，所以时间局部性的数量也很有限。

为了说明这个问题，系统设计者和编译器编写者一直在寻求一种称为预取（prefetching）的策略。其思想是通过预计在最近的将来要访问哪些块，然后使用特殊的机器指令事先取出这些块，放到高速缓存中，从而隐藏高速缓存不命中的等待时间。如果预取能做得很完美，那么每个块都会在程序想要引用它之前被拷贝到高速缓存中，因此每个加载指令都会命中。不过预取也有冒险性。因为预取流量与从 I/O 设备到主存的 DMA 流量共用总线，过多的预取会干扰 DMA 流量，减慢整个系统的性能。另一个潜在的问题是每个预取的高速缓存块都会驱逐一个现存的块。如果我们进行太多的预取，我们就要冒污染高速缓存（polluting the cache）的风险，有可能驱逐出前一次预取的，而程序还没有引用、不久的将来会引用的那个块。

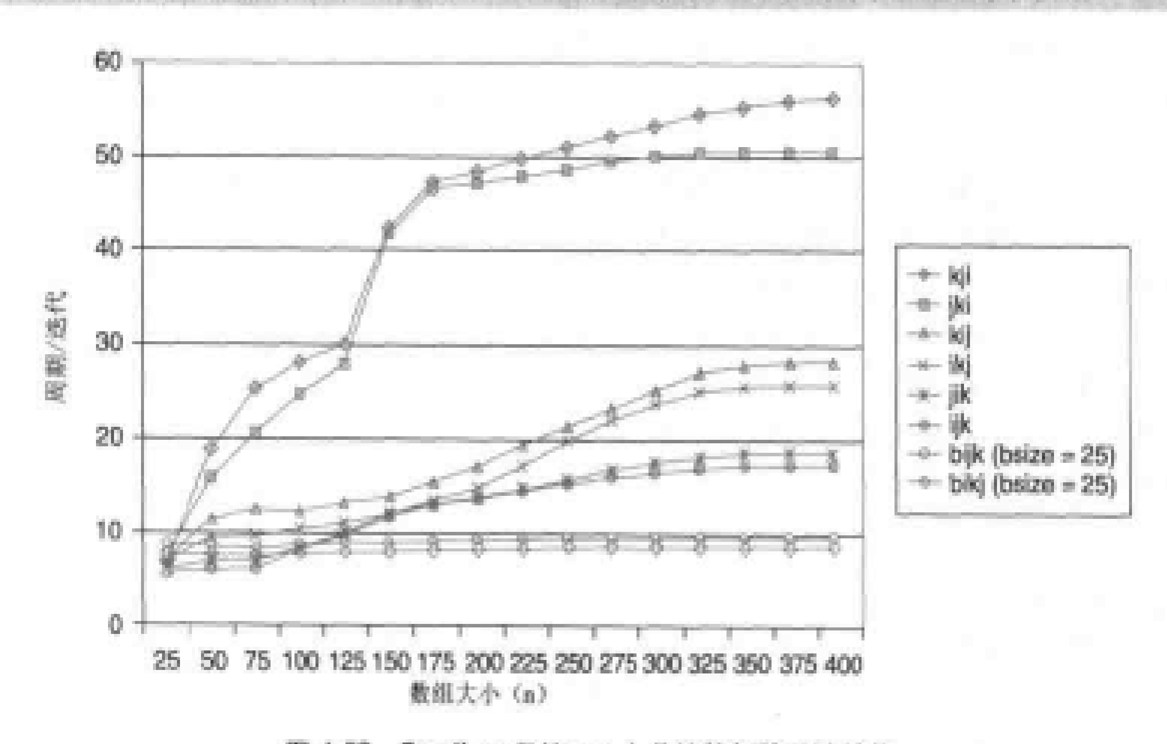


图 6.50 Pentium III Xeon 上分块的矩阵乘法性能

图例：bijk 和 bikj：分块的矩阵乘法的两个不同版本。同时也给出了图 6.47 中的非分块版本的性能，以供参考。

6.7 综合：利用程序中的局部性

正如我们看到的，存储系统被组织成一个存储设备的层次结构，较小、较快的设备靠近顶部，较大、较慢的设备靠近底部。由于这种层次结构，程序访问存储位置的有效速率不是一个数字能描述的。相反，它是一个变化很大的程序局部性的函数（我们称之为存储器山），变化可以有几个数量级。有良好局部性的程序从快速L1和L2高速缓存存储器中访问它的大部分数据。局部性差的程序从相对慢速的DRAM主存中访问它的大部分数据。

理解存储器层次结构本质的程序员能够利用这些知识，编写出更有效的程序，无论具体的存储系统结构是怎样的。特别地，我们推荐下列技术：

- 将你的注意力集中在内部循环上，大部分计算和存储器访问都发生在这里。
- 通过按照数据对象存储在存储器中的顺序来读数据，从而使得你程序中的空间局部性最大。
- 一旦从存储器中读入了一个数据对象，就尽可能多地使用它，从而使得你程序中的时间局部性最大。
- 记住，不命中率只是确定你代码性能的一个因素（虽然是重要的）。存储器访问数量也扮演着重要角色，有时需要在两者之间做一下折中。

6.8 小结

基本存储技术包括RAM（随机存储器）、ROM（非易失性存储器）和磁盘。RAM有两种基本类型。SRAM（静态RAM）快一些，但是也贵一些，它既可以用做CPU芯片上的高速缓存，也可以用做芯片外的高速缓存。动态RAM（DRAM）慢一点，也便宜一些，用做主存和图形帧缓冲区。非易失性存储器，也称为只读存储器（ROM），即使是在关电的时候，也能保持它们的信息，它们用来存储固件（firmware）。磁盘是非易失性存储设备，以每个位很低的成本保存大量的数据，代价是较长的访问时间。

一般而言，较快的存储技术每个位会更贵，而且容量较小。这些技术的价格和性能属性正在动态地以不同的速度变化着。特别地，DRAM和磁盘访问时间滞后于CPU周期时间。系统通过将存储器组织成存储设备的层次结构来弥补这些差异，在这个层次结构中，较小、较快的设备在顶部，较大、较慢的设备在底部。因为编写良好的程序有好的局部性，大多数数据都可以从较高层[3]得到服务，结果就是存储系统能以较高层的速度运行，但却有较低层的成本和容量。

程序员可以通过编写有良好空间和时间局部性的程序来动态地改进程序的运行时间。利用基于SRAM的高速缓存存储器特别重要，主要从L1高速缓存取数据的程序能比主要从存储器取数据的程序运行得快过一个数量级。

参考文献说明

存储器和磁盘技术变化得很快。根据我们的经验，最好的技术信息来源是制造商维护的Web页面。像Micron、Toshiba、Hyundai、Samsung、Hitachi和Kingston Technology这样的公司，提供了丰富的当前有关存储设备的技术信息。IBM、Maxtor和Seagate的页面也提供了类似的有关磁盘的

3 指存储器山中的层次。——译者

有用信息。

关于电路和逻辑设计的教科书提供了关于存储技术的详细信息[39，62]。IEEE Spectrum 出版了一系列对 DRAM 的概述文章[36]。计算机体系结构国际会议（ISCA）是一个关于 DRAM 存储性能特性的公共论坛[22，23]。

Wilkes 写了第一篇关于高速缓存存储器的论文[87]。Smith 写了一篇经典的综述[72]。Przybylski 编写了一本关于高速缓存设计的权威著作[59]。Hennessy 和 Patterson 提供了对高速缓存设计问题的全面讨论[33]。

Stricker 在[82]中介绍了存储器山的思想，作为对存储器系统的全面描述，并且在后来的工作描述中提出了术语“存储器山”。编译器研究者通过自动执行我们在 6.6 节中讨论过的那些手工代码转换，来增加局部性[14，25，45，48，54，60，89]。Carter 和同事们提出了一个可知晓高速缓存的存储控制器（a cache-aware memory controller）[11]。Seward 开发了一个开放源代码的高速缓存剖析程序，称为 cacheprof，它描述了 C 程序在任意模拟的高速缓存上的不命中行为（www.cacheprof.org）。

关于构造和使用磁盘存储设备也有大量的论著。许多存储技术研究者找寻方法，将单个的磁盘集合成更大、更健壮和更安全的存储池[12，28，29，57，90]。其他研究者找寻使用高速缓存和局部性来改进磁盘访问性能的方法[6，13]。像 Exokernel 这样的系统提供了更多的对磁盘和存储器资源的用户级控制[38]。像安德鲁文件系统[53]和 Coda[67]这样的系统，将存储器层次结构扩展到了计算机网络和移动笔记本电脑。Schindler 和 Ganger 开发了一个有趣的工具，它能自动描述 SCSI 磁盘驱动器的构造和性能[68]。

家庭作业

6.20 ◆◆

假设要求你设计一个每个磁道位数固定的磁盘。你知道每个磁道的位数是由最里层磁道的周长确定的，你可以假设它就是中间那个圆洞的周长。因此，如果你把磁盘中间的洞做得大一点，每个磁道的位数就会增大，但是总的磁道数会减少。如果用 r 来表示盘面的半径，$x \cdot r$ 表示圆洞的半径，那么 x 取什么值能使这个磁盘的容量最大？

6.21 ◆

下面的表给出了一些不同的高速缓存的参数。确定每个高速缓存的高速缓存组数量（S）、标记位数（t）、组索引位数（s）以及块偏移位数（b）。

高速缓存	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

6.22 ◆

这个问题是关于练习题 6.9 中的高速缓存的：

A. 列出所有会在组1中命中的十六进制存储器地址；

B. 列出所有会在组6中命中的十六进制存储器地址。

6.23 ◆◆

考虑下面的矩阵转置函数：

```
1   typedef int array[4][4];
2
3   void transpose2(array dst, array src)
4   {
5       int i, j;
6
7       for (i = 0; i < 4; i++) {
8           for (j = 0; j < 4; j++) {
9               dst[j][i] = src[i][j];
10          }
11      }
12  }
```

假设这段代码运行在一台具有如下属性的机器上：

- sizeof(int) = 4。
- 数组 src 从地址 0 开始，而数组 dst 从地址 64 开始（十进制）。
- 只有一个 L1 数据高速缓存，它是直接映射、直写、写分配的，块大小为 16 字节。
- 这个高速缓存总共有 32 个数据字节，初始为空。
- 对 src 和 dst 数组的访问分别是惟一的读和写不命中的来源。

对于每个 row 和 col，指明对 src[row][col]和 dst[row][col]的访问是命中（h）还是不命中（m）。例如，读 src[0][0]会不命中，而写 dst[0][0]也会不命中。

dst 数组				
	列 0	列 1	列 2	列 3
行 0	m			
行 1				
行 2				
行 3				

src 数组				
	列 0	列 1	列 2	列 3
行 0	m			
行 1				
行 2				
行 3				

6.24 ◆◆

对于一个总大小为 128 数据字节的高速缓存，重复练习题 6.23。

dst 数组				
	列 0	列 1	列 2	列 3
行 0	m			
行 1				
行 2				
行 3				

src 数组				
	列 0	列 1	列 2	列 3
行 0	m			
行 1				
行 2				
行 3				

6.25 ◆

3M 决定在白纸上印黄方格，做成小贴纸。这个过程中，他们需要设置方格中每个点的 CMYK（蓝色，红色，黄色，黑色）值。3M 雇佣你判定下面算法在一个 2048 字节、直接映射、块大小为 32 字节的数据高速缓存上的有效性。有如下定义：

```
1   struct point_color {
2       int c;
3       int m;
4       int y;
5       int k;
6   };
7
8   struct point_color square[16][16];
9   int i, j;
```

有如下假设：

- sizeof(int) == 4。
- square 起始于存储器地址 0。
- 高速缓存初始为空。
- 惟一的存储器访问是对于 square 数组中的元素。变量 i 和 j 被存放在寄存器中。

确定下列代码的高速缓存性能：

```
1   for (i = 0; i < 16; i++){
2       for (j = 0; j < 16; j++) {
3           square[i][j].c = 0;
```

```
4              square[i][j].m = 0;
5              square[i][j].y = 1;
6              square[i][j].k = 0;
7          }
8     }
```

A. 写总数是多少? ________

B. 在高速缓存中不命中的写总数是多少? ________

C. 不命中率是多少? ________

6.26 ◆

给定练习题 6.25 中的假设，确定下列代码的高速缓存性能:

```
1     for (i = 0; i < 16; i++){
2         for (j = 0; j < 16; j++) {
3             square[j][i].c = 0;
4             square[j][i].m = 0;
5             square[j][i].y = 1;
6             square[j][i].k = 0;
7         }
8     }
```

A. 写总数是多少? ________

B. 在高速缓存中不命中的写总数是多少? ________

C. 不命中率是多少? ________

6.27 ◆

给定练习题 6.25 中的假设，确定下列代码的高速缓存性能:

```
1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square[i][j].y = 1;
4         }
5     }
6     for (i = 0; i < 16; i++) {
7         for (j = 0; j < 16; j++) {
8             square[i][j].c = 0;
9             square[i][j].m = 0;
10            square[i][j].k = 0;
11        }
12    }
```

A. 写总数是多少? ________

B. 在高速缓存中不命中的写总数是多少? ________

C. 不命中率是多少? ________

6.28 ◆◆

你正在编写一个新的 3D 游戏，希望能名利双收。你现在正在写一个函数，在画下一帧之前先清空屏幕缓冲区。你现在工作的屏幕是 640×480 像素数组的。你工作的机器有一个 64KB 直接映射高速缓存，每行 4 个字节。你使用的 C 数据结构为：

```
1   struct pixel {
2       char r;
3       char g;
4       char b;
5       char a;
6   };
7
8   struct pixel buffer[480][640];
9   int i, j;
10  char *cptr;
11  int *iptr;
```

有如下假设：

- sizeof(char)==1 和 sizeof(int)==4。
- buffer 起始于存储器地址。
- 高速缓存初始为空。
- 惟一的存储器存问是对于 buffer 数组中的元素。变量 i、j、cptr 和 iptr 被存放在寄存器中。

下面代码中百分之多少的写会在高速缓存中不命中？

```
1   for (j = 0; j < 640; j++) {
2       for (i = 0; i < 480; i++){
3           buffer[i][j].r = 0;
4           buffer[i][j].g = 0;
5           buffer[i][j].b = 0;
6           buffer[i][j].a = 0;
7       }
8   }
```

6.29 ◆◆

给定练习题 6.28 中的假设，下面代码中百分之多少的写会在高速缓存中不命中？

```
1   char *cptr = (char *) buffer;
2   for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3       *cptr = 0;
```

6.30 ◆◆

给定练习题 6.28 中的假设，下面代码中百分之多少的写会在高速缓存中不命中？

```
1   int *iptr = (int *)buffer;
2   for (; iptr < ((int *)buffer + 640*480); iptr++)
3       *iptr = 0;
```

6.31 ◆◆◆

从 CS:APP 的网站上下载 mountain 程序，在你最喜欢的 PC/Linux 系统上运行它。根据结果估

计你系统上 L1 和 L2 高速缓存的大小。

6.32 ◆◆◆◆

在这项任务中，你会把你在第 5 章和第 6 章中学习到的概念应用到一个存储器使用频繁的应用的代码优化问题上。考虑一个拷贝并转置一个类型为 int 的 N×N 矩阵的过程。也就是，对于源矩阵 S 和目的矩阵 D，我们要将每个元素 $s_{i,j}$ 拷贝到 $d_{j,i}$。只用一个简单的循环就能实现这段代码：

```
1   void transpose(int *dst, int *src, int dim)
2   {
3       int i, j;
4
5       for (i = 0; i < dim; i++)
6           for (j = 0; j < dim; j++)
7               dst[j*dim + i] = src[i*dim + j];
8   }
```

这里，过程的参数是指向目的矩阵（dst）和源矩阵（src）的指针，以及矩阵的大小 N（dim）。要想使得这段代码运行得快，需要两种优化。首先，虽然函数在利用源矩阵的空间局部性上做得很好，但是它对大的 N 值的目的矩阵却做得很差。其次，GCC 产生的代码不是非常有效率。看看汇编代码，我们知道其中的循环需要 10 条指令，其中有 5 个会引用存储器——一个引用源矩阵，一个引用目的矩阵，而三个从栈中读局部变量。你的工作就是解决这些问题，设计一个运行得尽可能快的转置函数。

6.33 ◆◆◆◆

这项作业是练习题 6.32 的一个有趣的变体。考虑将一个有向图 g 转换成它对应的无向图 g′。图 g′有一条从顶点 u 到顶点 v 的边，当且仅当原图 g 中有一条 u 到 v 或者 v 到 u 的边。图 g 是由如下的它的邻接矩阵（adjacency matrix）G 表示的。如果 N 是 g 中顶点的数量，那么 G 是一个 N×N 的矩阵，它的元素是全 0 或者全 1。假设 g 的顶点是这样命名的：$v_0, v_1, \cdots, v_{N-1}$。那么如果有一条从 v_i 到 v_j 的边，那么 G[i][j]为 1，否则为 0。注意，邻接矩阵对角线上的元素总是 1，而无向图的邻接矩阵是对称的。只用一个简单的循环就能实现这段代码：

```
1   void col_convert(int *G, int dim) {
2       int i, j;
3
4       for (i = 0; i < dim; i++)
5           for (j = 0; j < dim; j++)
6               G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
7   }
```

你的工作是设计一个运行得尽可能快的函数。同前面一样，要提出一个好的解答，你需要应用你在第 5 章和第 6 章中所学到的概念。

练习题答案

练习题 6.1 答案

这里的思想是通过使纵横比 max(r,c)/min(r,c)最小，使得地址位数最小。换句话说，数组越接近

于正方形，地址位数越少。

组织	r	c	b_r	b_c	max(b_r, b_c)
16 × 1	4	4	2	2	2
16 × 4	4	4	2	2	2
128 × 8	16	8	4	3	4
512 × 4	32	16	5	4	5
1024 × 4	32	32	5	5	5

练习题 6.2 答案

这个小练习的主旨是确保你理解柱面和磁道之间的关系。一旦你弄明白了这个关系，那问题就很简单了：

$$\begin{aligned} \mathit{Disk\ capacity} &= \frac{512\ bytes}{sector} \times \frac{400\ sectors}{track} \times \frac{10\,000\ tracks}{surface} \times \frac{2\ surfaces}{platter} \times \frac{2\ platters}{disk} \\ &= 8\,192\,000\,000\ bytes \\ &= 8.192\ GB \end{aligned}$$

练习题 6.3 答案

对这个问题的解答是对磁盘访问时间公式的直接应用。平均旋转时间（以 ms 为单位）为

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ secs/15\,000\ RPM) \times 1\,000\ ms/sec \\ &\approx 2\ ms \end{aligned}$$

平均传送时间为

$$\begin{aligned} T_{avg\ transfer} &= (60 secs/15\,000\ RPM) \times 1/500\ sectors/track \times 1\,000\ ms/sec \\ &\approx 0.008\ ms \end{aligned}$$

总地来说，总的预计访问时间为

$$\begin{aligned} T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\ &= 8\ ms + 2\ ms + 0.008\ ms \\ &\approx 10\ ms \end{aligned}$$

练习题 6.4 答案

为了创建一个步长为 1 的引用模式，必须改变循环的次序，使得最右边的索引变化得最快：

```
1   int sumarray3d(int a[N][N][N])
2   {
3       int i, j, k, sum = 0;
4
5       for (k = 0; k < N; k++) {
```

```
6            for (i = 0; i < N; i++) {
7                for (j = 0; j < N; j++) {
8                    sum += a[k][i][j];
9                }
10           }
11       }
12       return sum;
13   }
```

这是一个很重要的思想。要保证你理解了为什么这种循环次序改变就能得到一个步长为1的访问模式。

练习题 6.5 答案

解决这个问题的关键在于想像出数组是如何在存储器中排列的，然后分析引用模式。函数clear1以步长为1的引用模式访问数组，因此明显地具有最好的空间局部性。函数clear2依次扫描N个结构中的每一个，这是好的，但是在每个结构中，它以步长不为1的模式跳到下列相对于结构起始位置的偏移处：0、12、4、16、8、20。所以clear2的空间局部性比clear1的要差。函数clear3不仅在每个结构中跳来跳去，而且还从结构跳到结构，所以clear3的空间局部性比clear2和clear1都要差。

练习题 6.6 答案

这个解答是对图6.26中各种高速缓存参数定义的直接应用。不那么令人兴奋，但是在你能真正理解高速缓存是如何工作的之前，你需要理解高速缓存的结构是如何导致这样划分地址位的。

	m	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1.	32	1024	4	1	256	22	8	2
2.	32	1024	8	4	32	24	5	3
3.	32	1024	32	32	1	27	0	5

练习题 6.7 答案

填充消除了冲突不命中。因此，四分之三的引用是命中的。

练习题 6.8 答案

有时候，理解为什么某种思想是不好的，能够帮助你理解为什么另一种是好的。这里，我们看到的坏的想法是用高位来索引高速缓存，而不是用中间的位。

A. 用高位做索引，每个连续的数组组块（chunk）是由2^t个块组成的，这里t是标记位数。因此，数组头2^t个连续的块都会映射到组0，接下来的2^t个块会映射到组1，依此类推。

B. 对于直接映射高速缓存(S,E,B,m) = (512,1,32,32)，高速缓存容量是512个32字节的块，每个高速缓存行中有t=18个标记位。因此，数组中头2^{18}个块会映射到组0，接下来2^{18}个块会映射到组1。因为我们的数组只由4096/32=512个块组成，所以数组中所有的块都被映射到组0。因此，在任何时刻，高速缓存至多只能保存一个数组块，即使数组足够小，能够完全放到高速缓存中。很明显，用高位做索引不能充分利用高速缓存。

练习题 6.9 答案

两个低位是块偏移（CO），然后是三位的组索引（CI），剩下的位作为标记（CT）：

12	11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

练习题 6.10 答案

地址：0x0E34

A. 地址格式（每个小格子表示一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. 存储器引用：

参数	值
高速缓存块偏移（CO）	0x0
高速缓存组索引（CI）	0x5
高速缓存标记（CT）	0x71
高速缓存命中吗？（Y/N）	Y
高速缓存返回的字节	0xB

练习题 6.11 答案

地址：0x0DD5

A. 地址格式（每个小格子表示一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	0	1	0	1
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. 存储器引用：

参　数	值
高速缓存块偏移（CO）	0x1
高速缓存组索引（CI）	0x5
高速缓存标记（CT）	0x6E
高速缓存命中吗？（Y/N）	N
高速缓存返回的字节	-

练习题 6.12 答案

地址：0x1FF4

A. 地址格式（每个小格子表示一个位）：

12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. 存储器引用：

参 数	值
高速缓存块偏移（CO）	0x0
高速缓存组索引（CI）	0x1
高速缓存标记（CT）	0xFF
高速缓存命中吗？（Y/N）	N
高速缓存返回的字节	

练习题 6.13 答案

这个问题是练习题 6.9~6.12 的一种逆过程，要求你反向工作，从高速缓存的内容推出会在某个组中命中的地址。在这种情况中，组 3 包含一个有效行，标记为 0x32。因为组中只有一个有效行，四个地址会命中。这些地址的二进制形式为 0 0110 0100 11xx。因此，在组 3 中命中的四个十六进制地址是：0x064C、0x064D、0x064E 和 0x064F。

练习题 6.14 答案

A. 解决这个问题的关键是想像出图 6.51 中的图像。注意，每个高速缓存行只包含数组的一个行，高速缓存正好只够保存一个数组，而且对于所有的 i、src 和 dst 的行映射到同一个高速缓存行。因为高速缓存不够大，不足以容纳这两个数组，所以对一个数组的引用总是驱逐出另一个数组的有用的行。例如，对 dst[0][0]写会驱逐当我们读 src[0][0]时加载进来的那一行，所以，当我们接下来读 src[0][1]时，我们会有一个不命中。

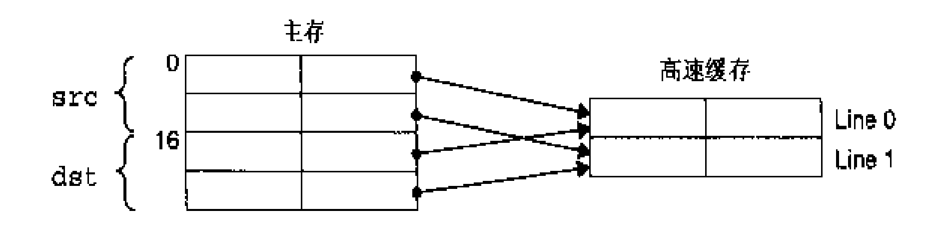


图 6.51 练习题 6.14 的图

dst 数组		
	列0	列 1
行 0	m	m
行 1	m	m

src 数组		
	列 0	列 1
行 0	m	m
行 1	m	h

B. 当高速缓存为 32 字节时，它足够大，能容纳这两个数组。因此，所有的不命中都是开始时的冷不命中。

dst 数组		
	列 0	列 1
行 0	m	h
行 1	m	h

src 数组		
	列 0	列
行 0	m	h
行 1	m	h

练习题 6.15 答案

每个 16 字节的高速缓存行包含着两个连续的 algae_position 结构。每个循环按照存储器顺序访问这些结构，每次读一个整数元素。所以，每个循环的模式就是不命中、命中、不命中、命中，依此类推。注意，对于这个问题，我们不必实际列举出读和不命中的总数，就能预测出不命中率。

A. 读总数是多少？512 个读。

B. 高速缓存不命中的读总数是多少？256 个不命中。

C. 不命中率是多少？256/512=50%。

练习题 6.16 答案

对这个问题的关键是注意到这个高速缓存只能保存数组的 1/2。所以，按照列顺序来扫描数组的第二部分会驱逐扫描第一部分时加载进来的那些行。例如，读 grid[16][0]的第一个元素会驱逐当我们读 grid[0][0]的元素时加载进来的那一行。这一行也包含 grid[0][1]。所以，当我们开始扫描下一列时，对 grid[0][1]第一个元素的引用会不命中，

A. 读总数是多少？512 个读。

B. 高速缓存不命中的读总数是多少？256 个不命中。

C. 不命中率是多少？256/512=50%。

D. 如果高速缓存有两倍大，那么不命中率会是多少呢？如果高速缓存有现在的两倍大，那么它能够保存整个 grid 数组。所有的不命中都会是开始时的冷不命中，而不命中率会是 1/4=25%。

练习题 6.17 答案

这个循环有很好的步长为 1 的引用模式，因此所有的不命中都是最开始时的冷不命中。

A. 读总数是多少？512 个读。

B. 高速缓存不命中的读总数是多少？128 个不命中。

C. 不命中率是多少？256/512=50%。

D. 如果高速缓存有两倍大，那么不命中率会是多少呢？无论高速缓存的大小增加多少，都不会改变不命中率，因为冷不命中是不可避免的。

练习题 6.18 答案

这个问题只是检查你是否理解了我们的讨论。步长对应于空间局部性，工作集大小对应于时间局部性。

练习题 6.19 答案

A. L1 的峰值吞吐率大约为 1000 MB/s，而时钟频率大约为 500 MHz。因此，访问 L1 中的一个字大约需要 500/1000×4 = 2 个周期。

B. 要估计 L2 的访问时间，我们需要确认存储器山上的一个区域，其中每个引用都在 L1 中不

命中，却在 L2 中命中。特别地，我们想要这样一个区域：①工作集对 L1 来说太大了，但却在 L2 的范围之内（例如，256 字节）；②步长超过了行的大小（例如，步长为 16 个字）。从存储器山的图中，可以观察到该区域（工作集大小=256，步长=16）中的有效吞吐率大约为 300 MB/s。因此，我们估计从 L2 中读一个字需要大约 500/300×4≈7 个周期。

要估计主存的访问时间，我们看看山上那个步长和工作集都最大的点，其中每个引用都在 L1 和 L2 中不命中。根据这幅图，这个区域（工作集大小=8M，步长=16）内的读吞吐率大约为 80 MB/s。因此，我们估计从主存中读出一个字大约需要 500/80×4≈25 个周期。

第 2 部分

在系统上运行程序

继续我们对计算机系统的探索，进一步来看看构建和运行程序的系统软件。链接器把我们程序的各个部分联合成一个单独的文件，处理器可以将这个文件加载到存储器（memory），并且执行它。现代操作系统与硬件合作，为每个程序提供一种幻像，好像这个程序是在独占地使用处理器和主存，而实际上，在任何时刻，系统上都有多个程序在运行。因此，要想在这样的系统上获得准确的测试值，就需要敏锐的洞察力和小心的设计规划。

在本书的第一部分，你很好地理解了程序和硬件之间的交互关系。本书的第二部分将拓宽你对系统的了解，使你牢固地掌握程序和操作系统之间的交互关系。你将学习到如何使用操作系统提供的服务来构建系统级程序，例如 Unix shell 和动态存储器分配包。

CHAPTER 7

链接

链接（linking）就是将不同部分的代码和数据收集和组合成为一个单一文件的过程，这个文件可被加载（或被拷贝）到存储器并执行。链接可以执行于编译时（compile time），也就是在源代码被翻译成机器代码时；也可以执行于加载时（load time），也就是在程序被加载器（loader）加载到存储器并执行时；甚至执行于运行时（run time），由应用程序来执行。在早期的计算机系统中，链接是手动执行的。在现代系统中，链接是由叫做链接器（linker）的程序自动执行的。

链接器在软件开发中扮演着一个关键的角色，因为它们使得分离编译（separate compilation）成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，我们只要简单地重新编译它，并将它重新链接到应用上，而不必重新编译其他文件。

链接通常是由链接器来安静地处理的，对于那些在编程入门课堂上构造小程序的学生而言，链接不是一个重要的议题。那为什么还要这么麻烦地学习关于链接的知识呢？

- 理解链接器将帮助你构造大型程序。构造大型程序的程序员经常会遇到由于缺少模块、缺少库或者不兼容的库版本引起的链接器错误。除非你理解链接器是如何解析引用、什么是库以及链接器是如何使用库来解析引用的，否则这类错误将令你感到迷惑和挫败。
- 理解链接器将帮助你避免一些危险的编程错误。Unix 链接器解析符号引用时所做的决定可以不动声色地影响你程序的正确性。在默认情况下，错误地定义多个全局变量的程序将通过链接器，而不产生任何警告信息。由此得到的程序会产生令人迷惑的运行时行为，而且非常难以调试。我们将向你展示这是如何发生的，以及该如何避免它。
- 理解链接将帮助你理解语言的作用域规则是如何实现的。例如，全局和局部变量之间的区别是什么？当你定义一个具有静态属性的变量或者函数时，到底实际意味着什么？
- 理解链接将帮助你理解其他重要的系统概念。链接器产生的可执行目标文件在重要的系统功能中扮演着关键角色，比如加载和运行程序、虚拟存储器、分页和存储器映射。
- 理解链接将使你能够开发共享库。多年以来，链接都被认为是相当简单和无趣的。然而，随着共享库和动态链接在现代操作系统中日益加强的重要性，链接成为了一个复杂的过程，它为知识丰富的程序员提供了强大的能力。比如，许多软件产品使用共享库在运行时来升级压缩包装的（shrink-wrapped）二进制程序。还有，大多数 Web 服务器都依赖于共享库的动态链接来提供动态内容。

这一章提供了关于链接各方面的一个彻底的讨论，从传统静态链接，到加载时的共享库的动态链接，以及到运行时的共享库的动态链接。我们将使用实际示例来描述基本的机制，而且我们将识别出链接问题在哪些情况中会影响你程序的性能和正确性。为了使描述具体和可理解，我们的讨论是基于这样的环境：一台 IA32 机器，上面运行着某个版本的 Unix，例如 Linux 或者 Solaris，使用的是标准的 ELF 目标文件格式。然而，无论是什么样的操作系统、ISA 或者是目标文件格式，基本的链接概念是通用的，认识到这一点是很重要的。细节可能不尽相同，但是概念是相同的。

7.1 编译器驱动程序

考虑图 7.1 中的 C 程序。它包含两个源文件：main.c 和 swap.c。函数 main()调用 swap，它交换外部全局数组 buf 中的两个元素。一般认为，这是一种奇怪的交换两个数字的方式，但是它将作为

贯穿本章的一个小的运行示例，来帮助我们说明关于链接是如何工作的一些重要知识点。

大多数编译系统提供编译驱动程序（compiler driver），它为用户，根据需求调用语言预处理器、编译器、汇编器和链接器。比如，要用 GNU 编译系统构造示例程序，我们就要通过在 shell 中输入下列命令行来调用 GCC 驱动程序：

```
unix > gcc -O2 -g -o p main.c swap.c
```

code/link/main.c

```
1   /* main.c */
2   void swap();
3
4   int buf[2] = {1, 2};
5
6   int main()
7   {
8       swap();
9       return 0;
10  }
```

code/link/main.c

（a）main.c

code/link/main.c

```
1   /* swap.c */
2   extern int buf[];
3
4   int *bufp0 = &buf[0];
5   int *bufp1;
6
7   void swap()
8   {
9       int temp;
10
11      bufp1 = &buf[1];
12      temp = *bufp0;
13      *bufp0 = *bufp1;
14      *bufp1 = temp;
15  }
```

code/link/main.c

（b）swap.c

图 7.1 示例程序 1

这个示例程序由两个源文件组成，main.c 和 swap.c。main 函数初始化一个两元素的整数数组，然后调用 swap 函数来交换这一对数。

图 7.2 概括了驱动程序在将示例程序从 ASCII 码源文件翻译成可执行目标文件时的行为。(如果你想自己看看这些步骤，用-v 选项来运行 GCC。）驱动程序首先运行 C 预处理器（cpp），它将 C 的源程序 main.c 翻译成一个 ASCII 码的中间文件 main.i：

```
cpp [other arguments] main.c /tmp/main.i
```

接下来，驱动程序运行 C 编译器（cc1），它将 main.i 翻译成一个 ASCII 汇编语言文件为 main.s。

```
cc1 /tmp/main.i main.c -O2 [other arguments] -o /tmp/main.s
```

然后，驱动程序运行汇编器（as），它将 main.s 翻译成一个可重定位目标文件（relocatable object file）main.o：

```
as [other arguments] -o /tmp/main.o /tmp/main.s
```

驱动程序经过相同的过程生成 swap.o。最后，它运行链接器程序 ld，将 main.o 和 swap.o 以及一些必要的系统目标文件组合起来，创建一个可执行的目标文件（executable object file）p：

```
ld -o p [system object files and args] /tmp/main.o /tmp/swap.o
```

要运行可执行文件 p，我们在 Unix shell 的命令行上输入它的名字：

```
unix> ./p
```

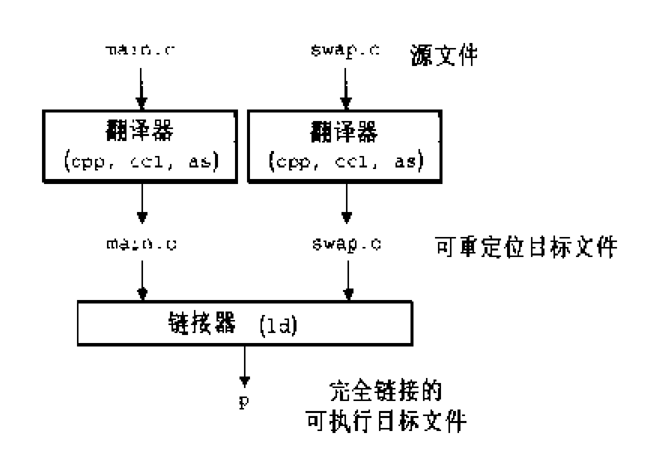


图 7.2 静态链接

链接器将可重定位目标文件组合成一个可执行目标文件p。

shell调用一个在操作系统中叫做*加载器*（loader）的函数，它拷贝可执行文件p中的代码和数据到存储器，然后将控制转移到这个程序的开头。

7.2 静态链接

像Unix ld程序这样的*静态链接器*（static linker）以一组可重定位目标文件和命令行参数作为输入，生成一个完全链接的可以加载和运行的可执行目标文件作为输出。输入的可重定位目标文件由各种不同的代码和数据节（section）组成。指令在一个节中，初始化的全局变量在另一个节中，而未初始化的变量又在另外一个节中。

为了创建可执行文件，链接器必须完成两个主要任务：

- *符号解析*（symbol resolution）。目标文件定义和引用符号。符号解析的目的是将每个符号引用和一个符号定义联系起来。
- *重定位*（relocation）。编译器和汇编器生成从地址零开始的代码和数据节。链接器通过把每个符号定义与一个存储器位置联系起来，然后修改所有对这些符号的引用，使得它们指向这个存储器位置，从而重定位这些节。

接下来的内容将更加详细地描述这些任务。在你阅读的时候，要记住关于链接器的一些基本事实：目标文件纯粹是字节块的集合。这些块中，有些包含程序代码，有些则包含程序数据，而其他的则包含指导链接器和加载器的数据结构。链接器将这些块连接起来，确定被链接块的运行时位置，并且修改代码和数据块中的各种位置。链接器对目标机器了解甚少，产生目标文件的编译器和汇编器已经完成了大部分工作。

7.3 目标文件

目标文件有三种形式：

- *可重定位目标文件*。包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件合并起来，创建一个可执行目标文件。

- 可执行目标文件。包含二进制代码和数据，其形式可以被直接拷贝到存储器并执行。
- 共享目标文件。一种特殊类型的可重定位目标文件，可以在加载或者运行时，被动态地加载到存储器并链接。

编译器和汇编器生成可重定位目标文件（包括共享目标文件）。链接器生成可执行目标文件。从技术上来说，一个目标模块（object module）就是一个字节序列，而一个目标文件（object file）就是一个存放在磁盘文件中的目标模块。不过，我们还是互换地使用这些术语。

各个系统之间，目标文件格式都不相同。第一个从贝尔实验室诞生的 Unix 系统使用的是 a.out 格式（直到今天，可执行文件仍然指的是 a.out 文件）。System V Unix 的早期版本使用的是 COFF（Common Object File format，一般目标文件格式）。Windows 使用的是 COFF 的一个变种，叫做 PE（Portable Executable，可移植可执行）格式。现代 Unix 系统——比如 Linux，还有 System V Unix 后来的版本，各种 BSD Unix，以及 SUN Solaris——使用的是 Unix ELF（Executable and Linkable Format，可执行和可链接格式）。尽管我们的讨论集中在 ELF 上，但是不管是哪种格式，基本的概念是相似的。

7.4 可重定位目标文件

图 7.3 展示了一个典型的 ELF 可重定位目标文件。ELF 头（ELF header）以一个 16 字节的序列开始，这个序列描述了字的大小和生成该文件的系统的字节顺序。ELF 头剩下的部分包含帮助链接器解析和解释目标文件的信息。其中包括 ELF 头的大小、目标文件的类型（比如，可重定位、可执行或者是共享的）、机器类型（比如，IA32）、节头部表（section header table）的文件偏移，以及节头部表中的表目大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的表目（entry）。

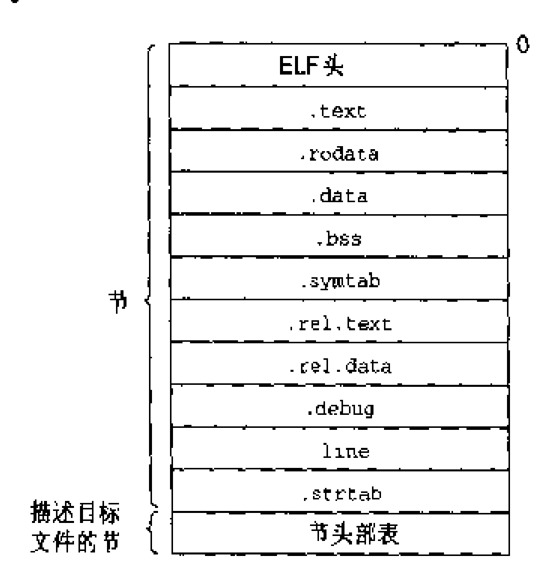


图 7.3 典型的 ELF 可重定位目标文件

夹在 ELF 头和节头部表之间的都是节。一个典型的 ELF 可重定位目标文件包含下面几个节：

.text:已编译程序的机器代码。

.rodata:只读数据，比如 printf 语句中的格式串和开关(switch)语句的跳转表(参见练习题 7.14)。

.data:已初始化的全局 C 变量。局部 C 变量在运行时被保存在栈中，既不出现在.data 节中，也不出现在.bss 节中。

.bss:未初始化的全局 C 变量。在目标文件中这个节不占据实际的空间，它仅仅是一个占位符。目标文件格式区分初始化和未初始化变量是为了空间效率：在目标文件中，未初始化变量不需要占据任何实际的磁盘空间。

.symtab:一个符号表（symbol table），它存放在程序中被定义和引用的函数和全局变量的信息。一些程序员错误地认为必须通过-g 选项来编译一个程序，得到符号表信息。实际上，每个可重定位目标文件在.symtab 中都有一张符号表。然而，和编译器中的符号表不同，.symtab 符号表不包含局部变量的表目。

.rel.text:当链接器把这个目标文件和其他文件结合时，.text 节中的许多位置都需要修改。一般而言，任何调用外部函数或者引用全局变量的指令都需要修改。另一方面，调用本地函数的指令则不需要修改。注意，可执行目标文件中并不需要重定位信息，因此通常省略，除非使用者显式地指示链接器包含这些信息。

.rel.data:被模块定义或引用的任何全局变量的信息。一般而言，任何已初始化全局变量的初始值是全局变量或者外部定义函数的地址都需要被修改。

.debug:一个调试符号表，其有些表目是程序中定义的局部变量和类型定义，有些表目是程序中定义和引用的全局变量，有些是原始的 C 源文件。只有以-g 选项调用编译驱动程序时，才会得到这张表。

.line:原始 C 源程序中的行号和.text 节中机器指令之间的映射。只有以-g 选项调用编译驱动程序时，才会得到这张表。

.strtab: 一个字符串表，其内容包括.symtab 和.debug 节中的符号表，以及节头部中的节名字。字符串表就是以 null 结尾的字符串序列。

旁注：为什么未初始化的数据称为.bss?

用术语.bss 来表示未初始化的数据是很普遍的。它起始于 IBM 704 汇编语言（大约在 1957 年）中“块存储开始（Block Storage Start）”指令的首字母缩写，并沿用至今。一个记住区分.data 和.bss 节的简单方法是把“bss”看成是“更好地节省空间（Better Save Space)!”的缩写。

7.5 符号和符号表

每个可重定位目标模块 m 都有一个符号表，它包含 m 所定义和引用的符号的信息。在链接器的上下文中，有三种不同的符号：

- 由 m 定义并能被其他模块引用的全局符号。全局链接器符号对应于非静态的 C 函数以及被定义为不带 C 的 static 属性的全局变量。
- 由其他模块定义并被模块 m 引用的全局符号。这些符号称为外部符号（external），对应于定义在其他模块中的 C 函数和变量。
- 只被模块 m 定义和引用的本地符号。有的本地链接器符号对应于带 static 属性的 C 函数和全局变量。这些符号在模块 m 中的任何地方都是可见的，但是不能被其他模块引用。目标

文件中对应于模块 m 的节和相应的源文件的名字也能获得本地符号。

认识到本地链接器符号和本地程序变量的不同是很重要的。.symtab 中的符号表不包含对应于本地非静态程序变量的任何符号。这些符号在运行时在栈中被管理，链接器对此类符号不感兴趣。

有趣的是，定义为带有 C static 属性的本地过程变量是不在栈中管理的。取而代之，编译器在.data 和.bss 中为每个定义分配空间，并在符号表中创建一个有惟一名字的本地链接器符号。比如，假设在同一模块中的两个函数定义了一个静态本地变量 x：

```
1    int f()
2    {
3        static int x = 0;
4        return x;
5    }
6
7    int g()
8    {
9        static int x = 1;
10       return x;
11   }
```

在这种情况中，编译器在.bss 中为两个整数分配空间，并引出（export）两个惟一的本地链接器符号给汇编器。比如，它可以用 x.1 表示函数 f 中的定义，而用 x.2 表示函数 g 中的定义。

给 C 语言初学者：利用 static 属性隐藏变量和函数名字

C 程序员使用 static 属性在模块内部隐藏变量和函数声明，就像你在 Java 和 C++中使用 public 和 private 声明一样。C 源代码文件扮演模块的角色。任何声明带有 static 属性的全局变量或者函数都是模块私有的。类似地，任何声明为不带 static 属性的全局变量和函数都是公共的，可以被其他模块访问。尽可能用 static 属性来保护你的变量和函数是很好的编程习惯。

符号表是由汇编器构造的，使用编译器输出到汇编语言.s 文件中的符号。.symtab 节中包含 ELF 符号表。这张符号表包含一个关于表目的数组。图 7.4 展示了每个表目（entry）的格式。

code/link/elfstructs.c

```
1   typedef struct {
2       int name;           /* string table offset */
3       int value;          /* section offset, or VM address */
4       int size;           /* object size in bytes */
5       char type:4,        /* data, func, section, or src file name (4 bits) */
6            binding:4;     /* local or global (4 bits) */
7       char reserved;      /* unused */
8       char section;       /* section header index, ABS, UNDEF, */
9                           /* or COMMON */
10  } Elf_Symbol;
```

code/link/elfstructs.c

图 7.4 ELF 符号表条目

type 和 binding 都是 4 位的。

name是字符串表中的字节偏移，指向符号的以null结尾的字符串名字。value是符号的地址。对于可重定位的模块来说，value是距定义目标的节的起始位置的偏移。对于可执行目标文件来说，该值是一个绝对运行时地址。size是目标的大小（以字节计算）。type通常要么是数据，要么是函数。符号表还可以包含各个节的表目，以及对应原始源文件的路径名的表目。所以这些目标的类型也有所不同。Binding域表示符号是本地的还是全局的。

每个符号都和目标文件的某个节相关联，由section域表示，该域也是一个到节头表的索引。有三个特殊的伪节（pseudosection），它们在节头表中是没有表目的：ABS代表不该被重定位的符号，UNDEF代表未定义的符号（比如，在本目标模块中引用，但是却在其他地方定义的符号），而COMMON表示还未被分配位置的未初始化的数据目标。对于COMMON符号，value域给出对齐请求，而size给出最小的大小。

比如，下面是main.o的符号表中的最后三个表目，通过GNU READELF工具显示出来。开始的8个表目没有显示出来，是链接器内部使用的本地符号。

```
Num:    Value  Size Type    Bind     Ot    Ndx Name
  8:        0     8 OBJECT  GLOBAL   0       3 buf
  9:        0    17 FUNC    GLOBAL   0       1 main
 10:        0     0 NOTYPE  GLOBAL   0     UND swap
```

在这个例子中，我们看到一个关于全局符号buf定义的表目，它是一个位于.data节中偏移为零（即value）处的8字节目标。其后跟随着的是全局符号main的定义，它是一个位于.text节中偏移为零处的17字节函数。最后一个表目来自对外部符号swap的引用。READELF通过一个整数索引来标识每个节。Ndx=1表示.text节，而Ndx=3表示.data节。

相似地，下面是swap.o的符号表表目：

```
Num:    Value  Size Type     Bind    Ot   Ndx Name
  8:        0     4 OBJECT   GLOBAL  0      3 bufp0
  9:        0     0 NOTYPE   GLOBAL  0    UND buf
 10:        0    39 FUNC     GLOBAL  0      1 swap
 11:        4     4 OBJECT   GLOBAL  0    COM bufp1
```

首先，我们看到一个关于全局符号bufp0定义的表目，它是从.data中偏移为零处开始的一个4字节的已初始化目标。下一个符号来自bufp0的初始化代码中的对外部符号buf的引用。后面紧随的是全局符号swap，它是一个位于.text中偏移为零处的39字节的函数。最后一个表目是全局符号bufp1，它是一个未初始化的4字节数据目标（要求4字节对齐），最终当这个模块被链接时它将作为一个.bss目标分配。

练习题 7.1

这个题目是关于图7.1（b）中的swap.o模块。对于每个在swap.o中定义或引用的符号，请指出它是否在模块swap.o中的.symtab节中有一个符号表表目。如果是，请指出定义该符号的模块（swap.o或者main.o）、符号类型（本地、全局或者外部）和它在模块中占据的节（.text、.data或者.bss）。

符号	swap.o.symtab 条目?	符号类型	在哪个模块中定义	节
buf				
bufp0				
bufp1				
swap				
temp				

7.6 符号解析

链接器解析符号引用的方法是将每个引用与它输入的可重定位目标文件的符号表中的一个确定的符号定义联系起来。对那些和引用定义在相同模块中的本地符号的引用，符号解析是非常简单明了的。编译器只允许每个模块中的每个本地符号只有一个定义。编译器还确保静态本地变量，它们也会有本地链接器符号，拥有惟一的名字。

不过，对全局符号的引用解析就棘手得多。当编译器遇到一个不是在当前模块中定义的符号（变量或函数名）时，它会假设该符号是在其他某个模块中定义的，生成一个链接器符号表表目，并把它交给链接器处理。如果链接器在它的任何输入模块中都找不到这个被引用的符号，它就输出一条（通常很难阅读的）错误信息并终止。比如，如果我们试着在一台 Linux 机器上编译和链接下面的源文件：

```
1    void foo(void);
2
3    int main() {
4        foo();
5        return 0;
6    }
```

那么编译器会没有障碍地运行，但是当链接器无法解析对 foo 的引用时，它会终止：

```
unix> gcc -Wall -O2 -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function 'main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to 'foo'
collect2: ld returned 1 exit status
```

对全局符号的符号解析很棘手，还因为相同的符号会被多个目标文件定义。在这种情况中，链接器必须要么标志一个错误，要么以某种方法选出一个定义并抛弃其他定义。Unix 系统采纳的方法包括编译器、汇编器和链接器之间的协作，这样也可能给不知情的程序员带来一些令人烦恼的问题。

旁注：对 C++和 Java 中链接器符号的毁坏（mangling）

C++和 Java 都允许重载方法，这些方法在源代码中有相同的名字，却有不同的参数列表。那么链接器是如何区别这些不同的重载函数之间的差异呢？C++和 Java 中能使用重载函数，是因为编译器将每个惟一的方法和参数列表组合编码成一个对链接器来说惟一的名字。这种编码过程叫做毁坏（mangling），而相反的过程叫做恢复（demangling）。

幸运的是，C++和 Java 使用兼容的毁坏策略。一个已毁坏类的名字是由名字中字符的整数数量，

后面跟原始名字组成的。比如，类 Foo 被编码成 3Foo。方法被编码为原始方法名，后面加上__，加上已毁坏类的类名，再加上每个参数的一个字母。比如，Foo::bar(int, long) 被编码为 bar__3Fooil。毁坏全局变量和模板名字的策略是相似的。

7.6.1 链接器如何解析多处定义的全局符号

在编译时，编译器输出每个全局符号给汇编器，或者是强（strong），或者是弱（weak），而汇编器把这个信息隐含地编码在可重定位目标文件的符号表里。函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号。对于图 7.1 中的示例程序，buf、bufp0、main 和 swap 是强符号，bufp1 是弱符号。

根据强弱符号的定义，Unix 链接器使用下面的规则来处理多处定义的符号：

- 规则 1：不允许有多个强符号。
- 规则 2：如果有一个强符号和多个弱符号，那么选择强符号。
- 规则 3：如果有多个弱符号，那么从这些弱符号中任意选择一个。

比如，假设我们试图编译和链接下面两个 C 模块：

```
1   /* foo1.c */
2   int main()
3   {
4       return 0;
5   }
```

```
1   /* bar1.c */
2   int main()
3   {
4       return 0;
5   }
```

在这个示例中，链接器将生成一条错误信息，因为强符号 main 被定义了多次（规则 1）：

```
unix> gcc foo1.c bar1.c
/tmp/cca015022.o: In function 'main':
/tmp/cca015022.o(.text+0x0): multiple definition of 'main'
/tmp/cca015021.o(.text+0x0): first defined here
```

相似地，链接器对于下面的模块也会生成一条错误信息，因为强符号 x 被定义了两次（规则 1）：

```
1   /* foo2.c */
2   int x = 15213;
3
4   int main()
5   {
6       return 0;
7   }
```

```
1   /* bar2.c */
2   int x = 15213;
3
4   void f()
5   {
6   }
```

然而，如果在一个模块里 x 未被初始化，那么链接器将安静地选择定义在另一个模块中的强符号（规则 2）：

```
1   /* foo3.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x = 15213;
```

```
1   /* bar3.c */
2   int x;
3
4   void f()
5   {
```

```
6                                    6        x = 15212;
7    int main()                      7    }
8    {
9        f();
10       printf("x = %d\n", x);
11       return 0;
12   }
```

在运行时，函数 f 将 x 的值由 15213 改为 15212，这会给 main 函数的作者带来不受欢迎的惊奇！注意，链接器通常不会表明它检测到多个 x 的定义：

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

如果 x 有两个弱定义，也会发生相同的事情（规则 3）：

```
1    /* foo4.c */                    1    /* bar4.c */
2    #include <stdio.h>              2    int x;
3    void f(void);                   3
4                                    4    void f()
5    int x;                          5    {
6                                    6        x = 15212;
7    int main()                      7    }
8    {
9        x = 15213;
10       f();
11       printf("x = %d\n", x);
12       return 0;
13   }
```

规则 2 和规则 3 的应用会造成一些不易察觉的运行时错误，对于不知情的程序员来说，是很难理解的，尤其是如果重复的符号定义还有不同的类型时。考虑下面这个例子，其中 x 在一个模块中定义为 int，而在另一个模块中定义为 double。

```
1    /* foo5.c */                    1    /* bar5.c */
2    #include <stdio.h>              2    double x;
3    void f(void);                   3
4                                    4    void f()
5    int x = 15213;                  5    {
6    int y = 15212;                  6        x = -0.0;
7                                    7    }
8    int main()
9    {
10       f();
11       printf("x = 0x%x y = 0x%x \n",
12              x, y);
```

```
13        return 0;
14    }
```

在一台IA32/Linux机器上，double类型是8个字节，而int类型是4个字节。因此，bar.c的第6行中的赋值x=-0.0将用负数的双精度浮点表示覆盖存储器中x和y的位置（foo5.c中的第5行和第6行）！

```
linux> gcc -o foobar5 foo5.c bar5.c
linux> ./foobar5
x = 0x0 y = 0x80000000
```

这是一个细微而令人讨厌的错误，尤其是因为它是默默发生的，编译系统不会给出警告，而且通常要在程序执行很久以后才表现出来，且远离错误的发生地。在一个拥有几百个模块的大型系统中，这种类型的错误相当难以修正，尤其因为许多程序员并不知道链接器是如何工作的。当你怀疑有此类错误时，带像GCC-warn-common这样的选项调用链接器，这个选项告诉链接器，在解析多定义的全局符号定义时，输出一条警告信息。

练习题 7.2

在此题中，REF(x.i)→DEF(x.k)表示链接器将把模块i中对符号x的任意引用与模块k中x的定义联系起来。对于下面的每个示例，用这种表示法来说明链接器将如何解析每个模块中的多个定义的符号。如果有一个链接时错误（规则1），输出“ERROR”。如果链接器从定义中任意选择一个（规则3），则输出“UNKNOWN”。

A.

```
/* Module 1 */                    /* Module 2 */
int main()                        int main;
{                                 int p2()
}                                 {
                                  }
(a) REF(main.1) --> DEF(_____._____)
(b) REF(main.2) --> DEF(_____._____)
```

B.

```
/* Module 1 */                    /* Module 2 */
void main()                       int main=1;
{                                 int p2()
}                                 {
                                  }
(a) REF(main.1) --> DEF(_____._____)
(b) REF(main.2) --> DEF(_____._____)
```

C.

```
/* Module 1 */                    /* Module 2 */
int x;                            double x=1.0;
void main()                       int p2()
{                                 {
}                                 }
(a) REF(x.1) --> DEF(_____._____)
(b) REF(x.2) --> DEF(_____._____)
```

7.6.2 与静态库链接

迄今为止，我们都是假设链接器读取一组可重定位目标文件，并把它们链接起来，成为一个输出的可执行文件。实际上，所有的编译系统都提供一种机制，将所有相关的目标模块打包为一个单独的文件，称为*静态库*（static library），它也可以用做链接器的输入。当链接器构造一个输出的可执行文件时，它只拷贝静态库里被应用程序引用的目标模块。

为什么系统要支持库的概念呢？以 ANSI C 为例，它定义了一组广泛的标准 I/O、串操作和整数算术函数，例如 atoi、printf、scanf 和 random。它们在 libc.a 库中，对每个 C 程序来说都是可用的。ANSI C 还在 libm.a 库中定义了一组广泛的浮点算术函数，例如 sin、cos 和 sqrt。

让我们来看看如果不使用静态库，编译器开发人员会使用什么方法来向用户提供这些函数。一种方法是让编译器辨认出对标准函数的调用，并直接生成相应的代码。Pascal，只提供了一小部分标准函数，采用的就是这种方法，但是这种方法对 C 而言是不合适的，因为 C 标准定义了大量的标准函数。这种方法将给编译器增加显著的复杂性，而且每次添加、删除或修改一个标准函数时，就需要一个新的编译器版本。然而，对于应用程序员而言，这种方法会是非常方便的，因为标准函数将总是可用的。

另一种方法是将所有的标准 C 函数都放在一个单独的可重定位目标模块中——比如说 libc.o 中——应用程序员可以把这个模块链接到他们的可执行文件中：

```
unix> gcc main.c /usr/lib/libc.o
```

这种方法的优点是它将编译器的实现与标准函数的实现分离开来，并且仍然对程序员保持适度的便利。然而，一个很大的缺点是系统中每个可执行文件现在都包含着一份标准函数集合的完全拷贝，这对磁盘空间是很大的浪费。（在一个典型的系统上，libc.a 大约是 8MB，而 libm.a 大约是 1MB。）更糟的是，每个正在运行的程序都将它自己的这些函数的拷贝放在存储器中，这又是极度浪费存储器的。另一个大的缺点是，对任何标准函数的任何改变，无论大小，都要求库的开发人员重新编译整个源文件，这是一个非常耗时的操作，使得标准函数的开发和维护变得很复杂。

我们通过为每个标准函数创建一个分离的可重定位文件，把它们存放在一个为大家所知的目录中来解决其中的一些问题。然而，这种方法要求应用程序员显式地链接合适的目标模块到它们的可执行文件中，这是一个容易出错而且耗时的过程：

```
unix> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

静态库概念被提出来，以解决这些不同方法的缺点。相关的函数可以被编译为独立的目标模块，然后封装成一个单独的静态库文件。然后，应用程序可以通过在命令行上指定单独的文件名字来使用这些在库中定义的函数。比如，使用标准 C 库和数学库中函数的程序可以用形式如下的命令行来编译和链接：

```
unix> gcc main.c /usr/lib/libm.a /usr/lib/libc.a ...
```

在链接时，链接器将只拷贝被程序引用的目标模块，这就减少了可执行文件在磁盘和存储器中的大小。另一方面，应用程序员只需要包含较少的库文件的名字（实际上，C 编译器驱动程序总是传送 libc.a 给链接器，所以前面提到的对 libc.a 的引用是不必要的）。

在 Unix 系统中，静态库以一种称为*存档*（archive）的特殊文件格式存放在磁盘中。存档文件是一组连接起来的可重定位目标文件的集合，有一个头部描述每个成员目标文件的大小和位置。存档

文件名由后缀.a 标识。为了使我们对库的讨论更加形象具体，假设我们想在一个叫做 libvector.a 的静态库中提供图 7.5 中的向量例程。

code/link/addvec.c

```
1   void addvec(int *x, int *y,
2               int *z, int n)
3   {
4       int i;
5
6       for (i = 0; i < n; i++)
7           z[i] = x[i] + y[i];
8   }
```

code/link/addvec.c

(a) addvec.o

code/link/multvec.c

```
1   void multvec(int *x, int *y,
2                int *z, int n)
3   {
4       int i;
5
6       for (i = 0; i < n; i++)
7           z[i] = x[i] * y[i];
8   }
```

code/link/ multvec.c

(b) multvec.o

图 7.5 libvector.a 中的成员目标文件

为了创建该库，我们将使用 AR 工具，如下：

```
unix> gcc -c addvec.c multvec.c
unix> ar rcs libvector.a addvec.o multvec.o
```

为了使用这个库，我们可以编写一个应用，比如图 7.6 中的 main.c，它调用 addvec 库例程（包含（或头）文件 vector.h 定义了 libvector.a 中例程的函数原型）。

code/link/main2.c

```
1   /* main2.c */
2   #include <stdio.h>
3   #include "vector.h"
4
5   int x[2] = {1, 2};
6   int y[2] = {3, 4};
7   int z[2];
8
9   int main()
10  {
11      addvec(x, y, z, 2);
12      printf("z = [%d %d]\n", z[0], z[1]);
13      return 0;
14  }
```

ocode/link/main2.c

图 7.6 示例程序 2

这个程序调用了静态 libvector.a 库中的成员函数。

为了创建这个可执行文件，我们将编译和链接输入文件 main.o 和 libvector.a：

```
unix> gcc -O2 -c main2.c
unix> gcc -static -o p2 main2.o ./libvector.a
```

图 7.7 概括了链接器的行为。-static 参数告诉编译器驱动程序，链接器应该构建一个完全链接的

可执行目标文件，它可以加载到存储器并运行，在加载时无须更进一步的链接了。当链接器运行时，它判定 addvec.o 定义的 addvec 符号是被 main.o 引用的，所以它拷贝 addvec.o 到可执行文件。因为程序不引用任何由 multvec.o 定义的符号，所以链接器就不会拷贝这个模块到可执行文件。链接器还会从 libc.a 拷贝 printf.o 模块，以及许多 C 运行时系统中的模块。

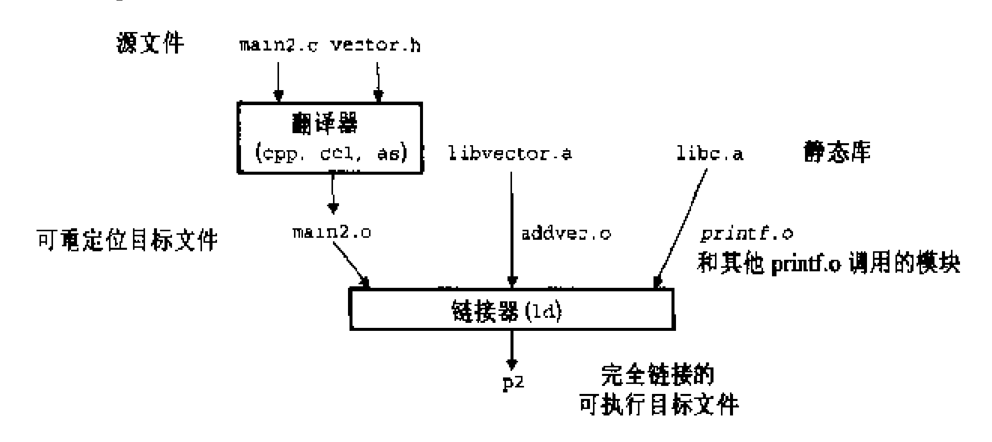


图 7.7 与静态库链接

7.6.3 链接器如何使用静态库来解析引用

虽然静态库是很有用而且重要的工具，但是它们同时也是程序员迷惑的源头，因为 Unix 链接器使用它们解析外部引用的方式是令人困惑的。在符号解析的阶段，链接器从左到右按照它们在编译器驱动程序命令行上出现的相同顺序来扫描可重定位目标文件和存档文件。(驱动程序自动将命令行中所有的.c 文件翻译为.o 文件。) 在这次扫描中，链接器维持一个可重定位目标文件的集合 E，这个集合中的文件会被合并起来形成可执行文件，和一个未解析的符号（也就是，引用了但是尚未定义的符号）集合 U，以及一个在前面输入文件中已定义的符号集合 D。初始地，E、U 和 D 都是空的。

- 对于命令行上的每个输入文件 f，链接器会判断 f 是一个目标文件还是一个存档文件（carchive)。如果 f 是一个目标文件，那么链接器把 f 添加到 E，修改 U 和 D 来反映 f 中的符号定义和引用，并继续下一个输入文件。
- 如果 f 是一个存档文件，那么链接器就尝试匹配 U 中未解析的符号和由存档文件成员定义的符号。如果某个存档文件成员 m，定义了一个符号来解析 U 中的一个引用，那么就将 m 加到 E 中，并且链接器修改 U 和 D 来反映 m 中的符号定义和引用。对存档文件中所有的成员目标文件都反复进行这个过程，直到 U 和 D 都不再发生变化。在此时，任何不包含在 E 中的成员目标文件都被丢弃，而链接器将继续到下一个输入文件。
- 如果当链接器完成对命令行上输入文件的扫描后，U 是非空的，那么链接器就会输出一个错误并终止。否则，它会合并和重定位 E 中的目标文件，从而构建输出的可执行文件。

不幸的是，这种算法会导致一些令人困扰的链接时错误，因为命令行上的库和目标文件的顺序非常重要。如果在命令行中，定义一个符号的库出现在引用这个符号的目标文件之前，那么引用就不能被解析，链接会失败。比如，考虑下面的命令行发生了什么？

```
unix> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function 'main':
```

```
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to 'addvec'
```

在处理 libvector.a 时，U 是空的，所以没有 libvector.a 中的成员目标文件会添加到 E 中。因此，对 addvec 的引用是绝不会被解析的，所以链接器会产生一条错误信息并终止。

关于库的一般准则是将它们放在命令行的结尾。如果各个库的成员是相互独立的——也就是说没有成员引用另一个成员定义的符号——那么这些库就可以以任何顺序放置在命令行的结尾处。

另一方面，如果库不是相互独立的，那么它们必须排序，使得对于每个被存档文件的成员外部引用的符号 s，在命令行中至少有一个 s 的定义是在对 s 的引用之后的。比如，假设 foo.c 调用 libx.a 和 libz.a 中的函数，而这两个库又调用 liby.a 中的函数。那么，在命令行中 libx.a 和 libz.a 必须处在 liby.a 之前：

```
unix> gcc foo.c libx.a libz.a liby.a
```

如果需要满足依赖需求，可以在命令行上重复库。比如，假设 foo.c 调用 libx.a 中的函数，该库又调用 liby.a 中的函数，而 liby.a 又调用 libx.a 中的函数。那么 libx.a 必须在命令行上重复出现：

```
unix> gcc foo.c libx.a liby.a libx.a
```

作为另一种方法，我们可以将 libx.a 和 liby.a 合并成一个单独的存档文件。

练习题 7.3

a 和 b 表示当前目录中的目标模块或者静态库，而 a→b 表示 a 依赖于 b，也就是说 b 定义了一个被 a 引用的符号。对于下面每种场景，请给出最小的命令行（也就是一个含有最少数量的目标文件和库参数的命令），使得静态链接器能解析所有的符号引用。

A. `p.o → libx.a`

B. `p.o → libx.a → liby.a`

C. `p.o → libx.a → liby.a` 且 `liby.a → libx.a → p.o`

7.7 重定位

一旦链接器完成了符号解析这一步，它就把代码中的每个符号引用和确定的一个符号定义（也就是，它的一个输入目标模块中的一个符号表表目）联系起来。在此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。现在就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时地址。重定位由两步组成：

- **重定位节和符号定义**。在这一步中，链接器将所有相同类型的节合并为同一类型的新的聚合节。例如，来自输入模块的.data 节被全部合并成一个节，这个节成为输出的可执行目标文件的.data 节。然后，链接器将运行时存储器地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。当这一步完成时，程序中的每个指令和全局变量都有惟一的运行时存储器地址了。
- **重定位节中的符号引用**。在这一步中，链接器修改代码节和数据节中对每个符号的引用，使得它们指向正确的运行时地址。为了执行这一步，链接器依赖于称为**重定位表目**（relocation entry）的可重定位目标模块中的数据结构，我们接下来将会描述这种数据结构。

7.7.1 重定位表目

当汇编器生成一个目标模块时，它并不知道数据和代码最终将存放在存储器中的什么位置。它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时汇编器遇到对最终位置未知的目标引用，它就会生成一个*重定位表目*（relocation entry），告诉链接器在将目标文件合并成可执行文件时如何修改这个引用。代码的重定位表目放在.relo.text 中。已初始化数据的重定位表目放在.relo.data 中。

图 7.8 展示了 ELF 重定位表目的格式。offset 是需要被修改的引用的节偏移。symbol 标识被修改引用应该指向的符号。type 告知链接器如何修改新的引用。

code/link/elfstructs.c

```
1   typedef struct {
2       int offset;       /* offset of the reference to relocate */
3       int symbol:24,    /* symbol the reference should point to */
4           type:8;       /* relocation type */
5   } Elf32_Rel;
```

code/link/elfstructs.c

图 7.8 ELF 重定位表目

每个表目表示一个必须重定位的引用。

ELF 定义了 11 种不同的重定位类型，有些相当隐秘。我们只关心其中两种最基本的重定位类型：

- R_386_PC32：重定位一个使用 32 位 PC 相关的地址引用。回想一下 3.6.3 节，一个 PC 相关地址就是距程序计数器（PC）的当前运行时值的偏移量。当 CPU 执行使用 PC 相关寻址的指令时，它就将在指令中编码的 32 位值加上 PC 的当前运行时值，得到有效地址（例如，call 指令的目标），PC 值通常是存储器中下一条指令的地址。
- R_386_32：重定位一个使用 32 位绝对地址的引用。通过绝对寻址，CPU 直接使用在指令中编码的 32 位值作为有效地址，不需要进一步修改。

7.7.2 重定位符号引用

图 7.9 展示了链接器的重定位算法的伪代码。

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset; /* ptr to reference to be relocated */
4
5           /* relocate a PC-relative reference */
6           if (r.type == R_386_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9           }
10
11          /* relocate an absolute reference */
12          if (r.type == R_386_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
```

```
14          }
15      }
```

图 7.9 重定位算法

第1行和第2行在每个节s以及与每个节相关联的重定位表目r上迭代执行。为了使描述具体化，我们假设每个节s是一个字节数组，每个重定位表目r是一个类型为Elf32_Rel的结构，如图7.8中的定义。另外，我们还假设当算法运行时，链接器已经为每个节和符号都选择了运行时地址（分别用ADDR(s)和ADDR(r.symbol)表示）。第3行计算的是需要重定位的4字节引用的数组s中的地址。如果这个引用使用的是PC相关寻址，那么它就用第5~9行来重定位。如果该引用使用的是绝对寻址，它就通过第11~13行来重定位。

重定位PC相关的引用

回想我们在图7.1（a）中的运行示例，main.o的.text节中的main程序调用swap程序，该程序是在swap.o中定义的。下面是call指令的反汇编列表，是由GNU OBJDUMP工具生成的：

```
6: e8 fc ff ff ff          call 7 <main+0x7>   swap();
                           7: R_386_PC32 swap  relocation entry
```

从这个列表中，我们看到call指令开始于节偏移0x6处，由1个字节的操作码0xe8和随后的32位引用0xfffffffc（十进制-4）组成，它是以小端法字节顺序存储的。我们还看到下一行显示的是这个引用的重定位表目。（回想一下，重定位表目和指令实际上是存放在目标文件的不同节中的。OBJDUMP工具为了方便将它们显示在一起。）重定位表目r由3个域组成：

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

这些域告诉链接器修改开始于偏移量0x7处的32位PC相关引用，使得在运行时它指向swap程序。现在，假设链接器已经判定：

```
ADDR(s) = ADDR(.text) = 0x80483b4
```

和

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8.
```

使用图7.9中的算法，链接器首先计算出引用的运行时地址（第7行）：

```
refaddr = ADDR(s)     + r.offset
        = 0x80483b4  + 0x7
        = 0x80483bb
```

然后，它将引用从当前值（-4）修改为0x9，使得它在运行时指向swap程序（第8行）：

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
        = (unsigned) (0x80483c8      + (-4)    - 0x80483bb)
        = (unsigned) (0x9)
```

在得到的可执行目标文件中，call指令有如下的重定位的形式：

```
80483ba: e8 09 00 00 00     call    80483c8 <swap>    swap();
```

在运行时，call 指令将存放在地址 0x80483ba 处。当 CPU 执行 call 指令时，PC 的值为 0x80483bf，即紧随在 call 指令之后的指令的地址。为了执行这条指令，CPU 执行以下的步骤：

```
1. push PC onto stack
2. PC <- PC + 0x9 = 0x80483bf + 0x9 = 0x80483c8
```

因此，要执行的下一条指令就是 swap 程序的第一条指令，这当然就是我们想要的！

你可能想知道为什么汇编器会生成 call 指令中的引用的初始值为-4。汇编器用这个值作为偏移量，是因为 PC 总是指向当前指令的下一条指令。在有不同指令大小和编码方式的不同的机器上，该机器的汇编器会使用不同的偏移量。这是一个很有用的技巧，它允许链接器透明地重定位引用，很幸运地不用知道某一台机器的指令编码。

重定位绝对引用

回想图 7.1 中我们的示例程序，swap.o 模块将全局指针 bufp0 初始化为指向全局数组 buf 的第一个元素的地址：

```
int *bufp0 = &buf[0];
```

因为 bufp0 是一个已初始化的数据目标，那么它将被存放在可重定位目标模块 swap.o 的.data 节中。因为它被初始化为一个全局数组的地址，所以它需要被重定位。下面是 swap.o 中.data 节的反汇编列表：

```
00000000 <bufp0>:
   0: 00 00 00 00                            int *bufp0 = &buf[0];
                          0: R_386_32 buf     relocation entry
```

我们看到.data 节包含一个 32 位引用，bufp0 指针，它的值为 0x0。重定位表目告诉链接器这是一个 32 位绝对引用，开始于偏移 0 处，必须重定位使得它指向符号 buf。现在，假设链接器已经判定：

```
ADDR(r.symbol) = ADDR(buf) = 0x8049454
```

链接器使用图 7.9 中算法的第 13 行修改了引用：

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr)
        = (unsigned) (0x8049454      + 0)
        = (unsigned) (0x8049454)
```

在得到的可执行目标文件中，引用有下面的重定位形式：

```
0804945c <bufp0>:
 804945c: 54 94 04 08                  Relocated!
```

总而言之，链接器在运行时确定，变量 bufp0 将放置在存储器地址 0x804945c 处，并且被初始化为 0x8049454，这个值就是 buf 数组的运行时地址。

swap.o 模块中的.text 节包含 5 个绝对引用，都以相似的方式进行重定位（参考练习题 7.12）。图 7.10 展示了最终的可执行目标文件中被重定位的.text 和.data 节。

code/link/p-exe.d

```
1   080483b4 <main>:
2    80483b4: 55                      push   %ebp
3    80483b5: 89 e5                   mov    %esp,%ebp
```

```
4    80483b7: 83 ec 08                sub    $0x8,%esp
5    80483ba: e8 09 00 00 00          call   80483c8 <swap>     swap();
6    80483bf: 31 c0                   xor    %eax,%eax
7    80483c1: 89 ec                   mov    %ebp,%esp
8    80483c3: 5d                      pop    %ebp
9    80483c4: c3                      ret
10   80483c5: 90                      nop
11   80483c6: 90                      nop
12   80483c7: 90                      nop

13  080483c8 <swap>:
14   80483c8: 55                      push   %ebp
15   80483c9: 8b 15 5c 94 04 08       mov    0x804945c,%edx      Get *bufp0
16   80483cf: a1 58 94 04 08          mov    0x8049458,%eax      Get buf[1]
17   80483d4: 89 e5                   mov    %esp,%ebp
18   80483d6: c7 05 48 95 04 08 58    movl   $0x8049458,0x8049548 bufp1 =
                                                                  &buf[1]
19   80483dd: 94 04 08
20   80483e0: 89 ec                   mov    %ebp,%esp
21   80483e2: 8b 0a                   mov    (%edx),%ecx
22   80483e4: 89 02                   mov    %eax,(%edx)
23   80483e6: a1 48 95 04 08          mov    0x8049548,%eax     Get *bufp1
24   80483eb: 89 08                   mov    %ecx,(%eax)
25   80483ed: 5d                      pop    %ebp
26   80483ee: c3                      ret
```

code/link/p-exe.d

(a) 已重定位的.text 节

code/link/pdata-exe-d.c

```
1   08049454 <buf>:
2    8049454: 01 00 00 00 02 00 00 00

3   0804945c <bufp0>:
4    804945c: 54 94 04 08                 Relocated!
```

code/link/pdata-exe-d.c

(b) 已重定位的.data 节

图 7.10 可执行文件 p 的已重定位的.text 和.data 节

原始的C代码在图 7.1 中。

练习题 7.4

本题是关于图 7.10 中的重定位程序的。

A. 第 5 行中对 swap 的重定位引用的十六进制地址是多少?

B. 第 5 行中对 swap 的重定位引用的十六进制值是多少?

C. 假设因为某种原因，链接器决定将.text 节放在 0x80483b8 处而不是 0x80483b4 处，那么这种情况下，第 5 行的重定位引用的十六进制值是多少?

7.8 可执行目标文件

我们已经看到链接器是如何将多个目标模块合并成一个可执行目标文件的。我们的 C 程序，开始时是一组 ASCII 文本文件，已经被转化为一个二进制文件，且这个二进制文件包含加载程序到存储器并运行它所需的所有信息。图 7.11 概括了一个典型的 ELF 可执行文件中的各类信息。

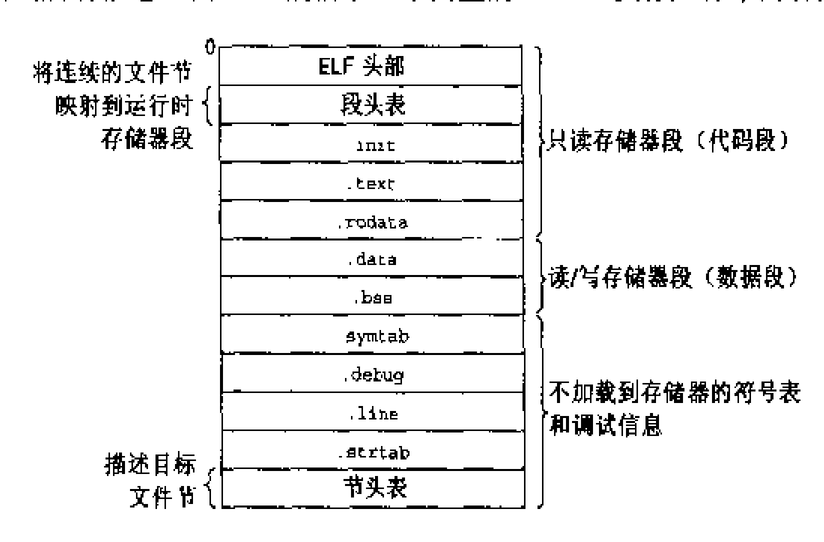


图 7.11 典型的 ELF 可执行目标文件

可执行目标文件的格式类似于可重定位目标文件的格式。ELF 头部描述文件的总体格式。它还包括程序的入口点（entry point），也就是当程序运行时要执行的第一条指令的地址。.text、.rodata 和.data 节和可重定位目标文件中的节是相似的，除了这些节已经被重定位到它们最终的运行时存储器地址以外。.init 节定义了一个小函数，叫做_init，程序的初始化代码会调用它。因为可执行文件是完全链接的（已被重定位了），所以它不再需要.relo 节。

ELF 可执行文件被设计为很容易加载到存储器，连续的可执行文件的组块（chunks）被映射到连续的存储器段。段头表（segment header table）描述了这种映射关系。图 7.12 展示了我们的示例可执行文件 p 的段头表，是由 OBJDUMP 显示的。

code/link/p-exe.d

```
   Read-only code segment
1  LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
2       filesz 0x00000448 memsz 0x00000448 flags r-x
   Read/write data segment
3  LOAD off    0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
4       filesz  0x000000e8 memsz 0x00000104 flags rw-
```

code/link/p-exe.d

图 7.12 示例可执行文件 p 的段头表

图注：off：文件偏移；vaddr/paddr：虚拟/物理地址；align：段对齐；filesz：目标文件中的段大小；memsz：存储器中的段大小；flags：运行时许可。

从段头表中，我们看到会根据可执行目标文件的内容初始化两个存储器段。第 1 行和第 2 行告诉我们第一个段（代码段）对齐到一个 4KB（2^{12}）的边界，有读/执行许可，开始于存储器地址

0x08048000处，总共的存储器大小是0x448字节，并且被初始化为可执行目标文件的头0x448个字节，其中包括ELF头部、段头表以及.init、.text和.rodata节。

第3行和第4行告诉我们第二个段（数据段）被对齐到一个4KB的边界，有读/写许可，开始于存储器地址0x08049448处，总的存储器大小为0x104字节，并用从文件偏移0x448处开始的0xe8个字节初始化，在此例中，偏移0x448处正是.data节的开始。该段中剩下的字节对应于运行时将被初始化为零的.bss数据。

7.9 加载可执行目标文件

要运行可执行目标文件p，我们可以在Unix shell的命令行中输入它的名字：

```
unix> ./p
```

因为p不是一个内置的shell命令，所以shell会认为p是一个可执行目标文件，通过调用某个驻留在存储器中称为加载器（loader）的操作系统代码来为我们运行之。任何Unix程序都可以通过调用execve函数来调用加载器，我们将在8.4.6节中详细地描述这个函数。加载器将可执行目标文件中的代码和数据从磁盘拷贝到存储器中，然后通过跳转到程序的第1条指令，即入口点（entry point），来运行该程序。这个将程序拷贝到存储器并运行的过程叫做*加载*（loading）。

每个Unix程序都有一个运行时存储器映像，如图7.13所示。在Linux系统中，代码段总是从地址0x08048000处开始。数据段是在接下来的下一个4KB对齐的地址处。运行时堆在接下来的读/写段之后的第一个4KB对齐的地址处，并通过调用malloc库往上增长。（我们将在10.9节中详细描述malloc和堆。）开始于地址0x40000000处的段是为共享库保留的。用户栈总是从地址0xbfffffff处开始，并向下增长的（向低存储器地址方向增长）。从栈的上部开始于地址0xc0000000处的段是为操作系统驻留存储器的部分（也就是内核）的代码和数据保留的。

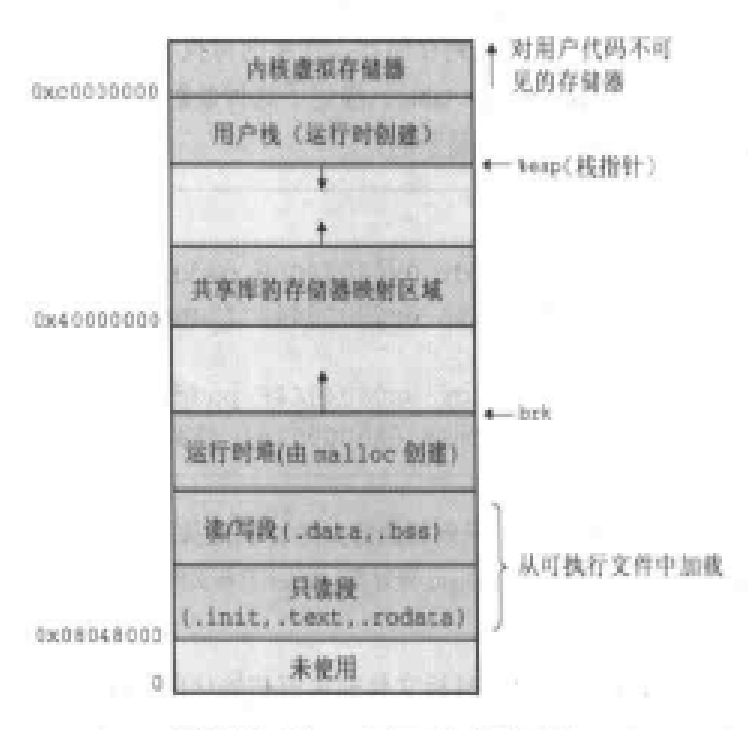


图7.13 Linux运行时存储器映像

当加载器运行时，它创建如图 7.13 所示的存储器映像。在可执行文件中段头表的指导下，加载器将可执行文件的相关内容拷贝到代码和数据段。接下来，加载器跳转到程序的入口点，也就是符号_start 的地址。在_start 地址处的启动代码（startup code）是在目标文件 crt1.o 中定义的，对所有的 C 程序都是一样的。图 7.14 展示了启动代码中特殊的调用序列。在从.text 和.init 节中调用了初始化例程后，启动代码调用 atexit 例程，这个程序附加了一系列在应用调用 exit 函数时应该调用的程序。exit 函数运行 atexit 注册的函数，然后通过调用_exit 将控制返回给操作系统。接着，启动代码调用应用程序的 main 程序，这就开始执行我们的 C 代码了。在应用程序返回之后，启动代码调用_exit 程序，它将控制返回给操作系统。

```
1   0x080480c0 <_start>:             /* entry point in .text */
2       call _libc_init_first        /* startup code in .text */
3       call _init                   /* startup code in .init */
4       call atexit                  /* startup code in .text */
5       call main                    /* application main routine */
6       call _exit                   /* returns control to OS */
7   /* control never reaches here */
```

图 7.14 在每个 C 程序中 crt1.o 启动例程的伪代码

注意：没有显示将每个函数的参数压入栈中的代码。

旁注：加载器实际上是如何工作的？

我们对于加载的描述从概念上来说是准确的，但也不是完全准确。为了理解加载实际是如何工作的，你必须理解进程、虚拟存储器和存储器映射的概念，这些我们还没有加以讨论。当我们在后面第 8 章和第 10 章中遇到这些概念时，我们将重新回到加载的问题上，并逐渐向你揭开它的神秘面纱。

对于不够有耐心的读者，下面是关于加载实际是如何工作的一个概述：Unix 系统中的每个程序都运行在一个进程上下文中，这个进程上下文有自己的虚拟地址空间。当 shell 运行一个程序时，父 shell 进程生成一个子进程，它是父进程的一个复制品。子进程通过 execve 系统调用启动加载器。加载器删除子进程已有的虚拟存储器段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零。通过将虚拟地址空间中的页映射到可执行文件的页大小的组块（chunks），新的代码和数据段被初始化为可执行文件的内容。最后，加载器跳转到_start 地址，它最终会调用应用的 main 函数。除了一些头部信息，在加载过程中没有任何从磁盘到存储器的数据拷贝。直到 CPU 引用一个被映射的虚拟页，才会进行拷贝，此时，操作系统利用它的页面调度机制自动将页面从磁盘传送到存储器。

练习题 7.5

A. 为什么每个 C 程序都需要一个叫做 main 的函数？

B. 你想过为什么 C 的 main 函数可以通过调用 exit 或者执行一条 return 语句，或者两者都不做，而程序仍然可以正确终止吗？请解释。

7.10 动态链接共享库

我们在 7.6.2 节中研究的静态库针对的许多问题是应用程序如何使用大量可用的相关函数。然

而，静态库仍然有一些明显的缺点。静态库和所有的软件一样，需要定期维护和更新。如果应用程序员想要使用一个库的最新版本，他们必须以某种方式了解到该库的更新情况，然后显式地将他们的程序与新的库重新链接。

另一个问题是几乎每个C程序都使用标准I/O函数，比如printf和scanf。在运行时，这些函数的代码会被复制到每个运行进程的文本段中。在一个运行50～100个进程的典型系统上，这会是对稀少的存储器系统资源的极大浪费。（存储器的一个有趣属性就是不论一个系统中有多大的存储器，它总是一种稀有的资源。磁盘空间和厨房的垃圾桶同样有这种属性。）

共享库（shared library）是致力于解决静态库缺陷的一个现代创新产物。共享库是一个目标模块，在运行时，可以加载到任意的存储器地址，并在存储器中和一个程序链接起来。这个过程称为**动态链接**（dynamic linking），是由一个叫做**动态链接器**（dynamic linker）的程序来执行的。

共享库也称为**共享目标**（shared object），在Unix系统中通常用.so后缀来表示。微软的操作系统大量地利用了共享库，它们称为DLL（动态链接库）。

共享库的"共享"在两个方面有所不同。首先，在任何给定的文件系统中，对于一个库只有一个.so文件。所有引用该库的可执行目标文件共享这个.so文件中的代码和数据，而不是像静态库的内容那样被拷贝和嵌入到引用它们的可执行的文件中。其次，在存储器中，一个共享库的.text节只有一个副本可以被不同的正在运行的进程共享。在第10章我们学习虚拟存储器时将更加详细地讨论这个问题。

图7.15概括了图7.6中示例程序的动态链接过程。为了构造图7.5中向量运算示例程序的共享库libvector.so，我们会调用编译器，给链接器如下特殊指令：

```
unix> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

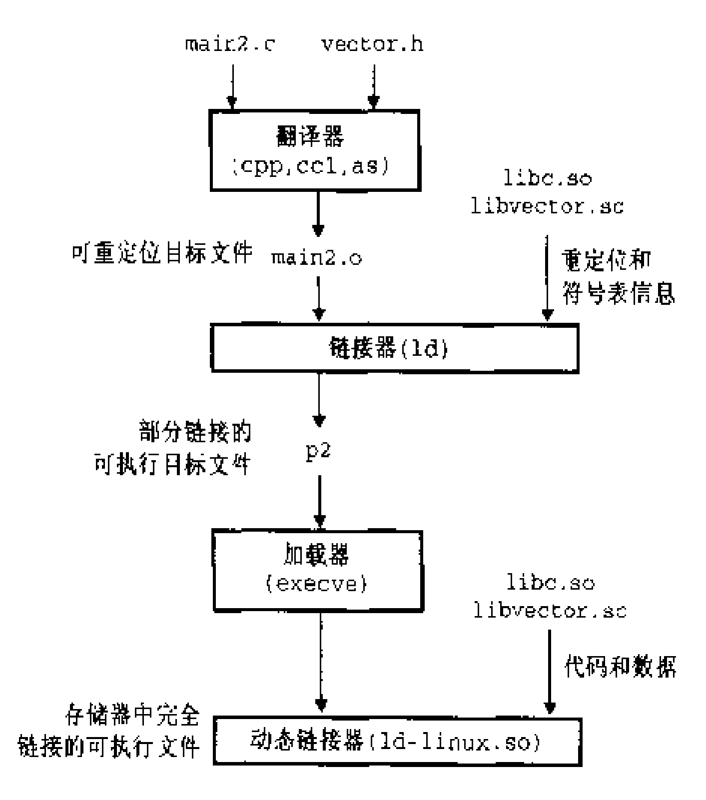


图7.15 用共享库来动态链接

-fPIC 选项指示编译器生成与位置无关的代码（下一节将详细讨论这个问题）。-shared 选项指示链接器创建一个共享的目标文件。

一旦我们创建了这个库，我们随后就要将它链接到图 7.6 的示例程序中。

```
unix> gcc -o p2 main2.c ./libvector.so
```

这样就创建了一个可执行目标文件 p2，而此文件的形式使得它在运行时可以和 libvector.so 链接。基本的思路是当创建可执行文件时，静态执行一些链接，然后在程序加载时，动态完成链接过程。

认识到这一点是很重要的：在此时刻，没有任何 libvector.so 的代码和数据节被真的拷贝到可执行文件 p2 中。取而代之的是，链接器拷贝了一些重定位和符号表信息，它们使得运行时可以解析对 libvector.so 中代码和数据的引用。

当加载器加载和运行可执行文件 p2 时，它利用 7.9 节中讨论过的技术，加载部分链接的可执行文件 p2。接着，它注意到 p2 包含一个.interp 节，这个节包含动态链接器的路径名，动态链接器本身就是一个共享目标（比如，在 Linux 系统上的 LD-LINUX.SO）。加载器不再像它通常那样将控制传递给应用，取而代之的是加载和运行这个动态链接器。

然后，动态链接器通过执行下面的重定位完成链接任务：

- 重定位 libc.so 的文本和数据到某个存储器段。在 IA32/Linux 系统中，共享库被加载到从地址 0x40000000 开始的区域中（参见图 7.13）。
- 重定位 libvector.so 的文本和数据到另一个存储器段。
- 重定位 p2 中所有对由 libc.so 和 libvector.so 定义的符号的引用。

最后，动态链接器将控制传递给应用程序。从这个时刻开始，共享库的位置就固定了，并且在程序执行的过程中都不会改变。

7.11 从应用程序中加载和链接共享库

到此刻为止，我们已经讨论了在应用程序执行之前，即应用程序被加载时，动态链接器加载和链接共享库的情景。然而，应用程序还可能在它运行时要求动态链接器加载和链接任意共享库，而无需在编译时链接那些库到应用中。

动态链接是一项强大有用的技术。下面是一些现实世界中的例子：

- 分发软件。微软 Windows 应用的开发者常常利用共享库来分发软件更新。他们生成一个共享库的新版本，然后用户可以下载，并用它替代当前的版本。下一次他们运行应用程序时，应用将自动链接和加载新的共享库。
- 构建高性能 Web 服务器。许多 Web 服务器生成动态内容，比如个性化的 Web 页面、账户余额和广告标语。早期的 Web 服务器通过使用 fork 和 execve 创建一个子进程，并在该子进程的上下文中运行 CGI 程序，来生成动态内容。然而，现代高性能的 Web 服务器可以使用基于动态链接的更有效和完善的方法来生成动态内容。

  其思路是将生成动态内容的每个函数打包在共享库中。当一个来自 Web 浏览器的请求到达时，服务器动态地加载和链接适当的函数，然后直接调用它，而不是使用 fork 和 execve 在子进程的上下文中运行函数。函数会一直缓存在服务器的地址空间中，所以只要一个简

单的函数调用的开销就可以处理随后的请求了。这对一个繁忙的网站来说是有很大影响的。

更进一步，可以在运行时，无需停止服务器，更新已存在的函数，以及添加新的函数。

像 Linux 和 Solaris 这样的 Unix 系统，为动态链接器提供了一个简单的接口，允许应用程序在运行时加载和链接共享库。

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
```

返回：若成功则为指向句柄的指针，若出错则为 Null。

dlopen 函数加载和链接共享库 filename。用以前带 RTLD_GLOBAL 选项打开的库解析 filename 中的外部符号。如果当前可执行文件是带-rdynamic 选项编译的，那么对符号解析而言，它的全局符号也是可用的。flag 参数必须要么包括 RTLD_NOW，该标志告诉链接器立即解析对外部符号的引用，要么包括 RTLD_LAZY 标志，该标志指示链接器推迟符号解析直到执行来自库中的代码时。这两个值中的任意一个都可以和 RTLD_GLOBAL 标志取或。

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
```

返回：若成功则为指向符号的指针，若出错则为 Null。

dlsym 函数的输入是一个指向前面已经打开共享库的句柄和一个符号名字，如果该符号存在，就返回符号的地址，否则返回 NULL。

```
#include <dlfcn.h>

int dlclose (void *handle);
```

返回：若成功则为 0，若出错则为 1。

如果没有其他共享库还在使用这个共享库，dlclose 函数就卸载该共享库。

```
#include <dlfcn.h>

const char *dlerror(void);
```

返回：如果前面对 dlopen、dlsym 或 dlclose 的调用失败，则为错误消息，如果前面的调用成功，则为 Null。

dlerror 函数返回一个字符串，它描述的是调用 dlopen、dlsym 或者 dlclose 函数时发生的最近的错误，如果没有错误发生，就返回 NULL。

图 7.16 展示了我们如何利用这个接口动态链接我们的 libvector.so 共享库（图 7.5），然后调用它的 addvec 程序。要编译这个程序，我们将以下面的方式调用 GCC：

```
unix> gcc -rdynamic -O2 -o p3 main3.c -ldl
```

code/link/dll.c

```
1   #include <stdio.h>
2   #include <dlfcn.h>
3
4   int x[2] = {1, 2};
```

```
5   int y[2] = {3, 4};
6   int z[2];
7
8   int main()
9   {
10      void *handle;
11      void (*addvec)(int *, int *, int *, int);
12      char *error;
13
14      /* dynamically load the shared library that contains addvec() */
15      handle = dlopen("./libvector.so", RTLD_LAZY);
16      if (!handle) {
17          fprintf(stderr, "%s\n", dlerror());
18          exit(1);
19      }
20
21      /* get a pointer to the addvec() function we just loaded */
22      addvec = dlsym(handle, "addvec");
23      if ((error = dlerror()) != NULL) {
24          fprintf(stderr, "%s\n", error);
25          exit(1);
26      }
27
28      /* Now we can call addvec() just like any other function */
29      addvec(x, y, z, 2);
30      printf("z = [%d %d]\n", z[0], z[1]);
31
32      /* unload the shared library */
33      if (dlclose(handle) < 0) {
34          fprintf(stderr, "%s\n", dlerror());
35          exit(1);
36      }
37      return 0;
38  }
```
code/link/dll.c

图 7.16 一个动态加载和链接共享库 libvector.so 的应用程序

旁注：共享库和 Java 本地接口

Java 定义了一个标准调用规则，叫做 Java 本地接口（Java Native Interface，JNI），它允许 Java 程序调用"本地的"C 和 C++函数。JNI 的基本思想是将本地 C 函数，比如说 foo，编译到共享库中，比如说 foo.so。当一个正在运行的 Java 程序试图调用函数 foo 时，Java 解释程序利用 dlopen 接口（或者某个类似于此的东西）动态链接和加载 foo.so，然后再调用 foo。

7.12 *与位置无关的代码（PIC）

共享库的一个主要目的就是允许多个正在运行的进程共享存储器中相同的库代码，因而节约宝

贵的存储器的资源。那么，多个进程是如何共享一个程序的一个拷贝的呢？一种方法是给每个共享库分配一个事先预备的专用的地址空间组块（chunk），然后要求加载器总是在这个地址加载共享库。虽然这种方法很简单，但是它也造成了一些严重的问题。首先，它对地址空间的使用效率不高，因为即使一个进程不使用这个库，那部分空间还是会被分配出来。第二，它也难以管理。我们将不得不保证没有组块会重叠。每次当一个库修改了之后，我们必须确认它的已分配的组块还适合它的大小。如果不适合了，我们必须找一个新的组块。并且，如果我们创建了一个新的库，我们还必须为它寻找空间。随着时间的进展，假设在一个系统中有了成百个库和各种版本的库，就很难避免地址空间分裂成大量小的、未使用而又不再能使用的小洞。甚至更糟的是，对每个系统而言，从库到存储器的分配都是不同的，这就引起了更多令人头痛的管理问题。

一种更好的方法是编译库代码，使得不需要链接器修改库代码，就可以在任何地址加载和执行这些代码。这样的代码叫做*与位置无关的代码*（position-independent code，PIC）。用户对GCC使用-fPIC选项指示GNU编译系统生成PIC代码。

在一个IA32系统中，对同一个目标模块中过程的调用是不需要特殊处理的，因为引用是PC相关的，已知偏移量，就已经是PIC了（参见练习题7.4）。然而，对外部定义的过程调用和对全局变量的引用通常不是PIC，因为它们都要求在链接时重定位。

7.12.1 PIC数据引用

编译器通过运用以下有趣的事实来生成对全局变量的PIC引用：无论我们在存储器中的何处加载一个目标模块（包括共享目标模块），数据段总是分配为紧随在代码段后面。因此，代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量，与代码段和数据段的绝对存储器位置是无关的。

为了运用这个事实，编译器在数据段开始的地方创建了一个表，叫做*全局偏移量表*（global offset table，GOT）。GOT包含每个被这个目标模块引用的全局数据目标的表目。编译器还为GOT中每个表目生成一个重定位记录。在加载时，动态链接器会重定位GOT中的每个表目，使得它包含正确的绝对地址。每个引用全局数据的目标模块都有一张自己的GOT。

在运行时，使用下面的代码形式，通过GOT间接地引用每个全局变量：

```
    call L1
L1: popl %ebx;              # ebx contains the current PC
    addl $VAROFF, %ebx      # ebx points to the GOT entry for var
    movl (%ebx), %eax       # reference indirect through the GOT
    movl (%eax), %eax
```

在这段代码中，对L1的调用将返回地址（正好就是popl指令的地址）压入栈中。随后，popl指令把这个地址弹出到%ebx中。这两条指令的最终效果是将PC的值移到寄存器%ebx中。

指令addl给%ebx增加一个常量偏移量，使得它指向GOT中适当的表目，该表目包含数据项的绝对地址。此时，就可以通过包含在%ebx中的GOT表目间接地引用全局变量了。在这个示例中，两条movl指令（间接地通过GOT）加载全局变量的内容到寄存器%eax中。

PIC代码有性能缺陷。现在每个全局变量引用需要五条指令而不是一条，还需要一个额外的对GOT的存储器引用。而且，PIC代码还要用一个额外的寄存器来保持GOT表目的地址。在具有大

寄存器文件的机器上，这不是一个大问题。然而，在寄存器供应不足的 IA32 系统中，即使失掉一个寄存器也会造成寄存器溢出到栈中。

7.12.2 PIC 函数调用

PIC 代码当然可以用相同的方法来解析外部过程调用：

```
    call L1
L1: popl %ebx;             # ebx contains the current PC
    addl $PROCOFF, %ebx    # ebx points to GOT entry for proc
    call *(%ebx)           # call indirect through the GOT
```

不过，这种方法对每一个运行时过程调用都要求三条额外的指令。取而代之，ELF 编译系统使用一种有趣的技术，叫做*延迟绑定*（lazy binding），将过程地址的绑定推迟到第一次调用该过程时。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的存储器引用。

延迟绑定是通过两个数据结构之间简洁但又有些复杂的交互来实现的，这两个数据结构是：GOT 和 PLT（procedure linkage table，*过程链接表*）。如果一个目标模块调用定义在共享库中的任何函数，那么它就有自己的 GOT 和 PLT。GOT 是.data 节的一部分，PLT 是.text 节的一部分。

图 7.17 展示了图 7.6 中示例程序 main.o 的 GOT 的格式。头三条 GOT 表目是特殊的：GOT[0]包含.dynamic 段的地址，这个段包含了动态链接器用来绑定过程地址的信息，比如符号表的位置和重定位信息；GOT[1]包含一些定义这个模块的信息；GOT[2]包含动态链接器的延迟绑定代码的入口点。

地址	表目	内容	描　述
08049674	GOT[0]	0804969c	.dynamic 节的地址
08049678	GOT[1]	4000a9f8	链接器的标识信息
0804967c	GOT[2]	4000596f	动态链接器中的入口点
08049680	GOT[3]	0804845a	PLT[1]中 pushl 地址（printf）
08049684	GOT[4]	0804846a	PLT[2]中 pushl 地址（addvec）

图 7.17　可执行文件 p2 的全局偏移量表（GOT）

原始代码见图 7.5 和图 7.6。

定义在共享目标中并被 main.o 调用的每个过程在 GOT 中都会有一个表目，从 GOT[3]表目开始。对于示例程序，我们给出了 printf 和 addvec 的 GOT 表目，printf 定义在 libc.so 中，而 addvec 定义在 libvector.so 中。

图 7.18 展示了我们示例程序 p2 的 PLT。PLT 是一个 16 字节表目的数组。第一个表目 PLT[0]是一个特殊表目，它跳转到动态链接器中。每个被调用的过程在 PLT 中都有一个表目，从 PLT[1]开始。在图中，PLT[1]对应于 printf，PLT[2]对应于 addvec。

```
PLT[0]
08048444: ff 35 78 96 04 08 pushl 0x8049678    # push &GOT[1]
 804844a: ff 25 7c 96 04 08 jmp   *0x804967c   # jmp to *GOT[2](linker)
 8048450: 00 00                                # padding
```

```
 8048452: 00 00                                          # padding

PLT[1] <printf>
 8048454: ff 25 80 96 04 08 jmp    *0x8049680    # jmp to *GOT[3]
 804845a: 68 00 00 00 00    pushl $0x0           # ID for printf
 804845f: e9 e0 ff ff ff    jmp    8048444       # jmp to PLT[0]

PLT[2] <addvec>
 8048464: ff 25 84 96 04 08 jmp    *0x8049684    # jump to *GOT[4]
 804846a: 68 08 00 00 00    pushl $0x8           # ID for addvec
 804846f: e9 d0 ff ff ff    jmp    8048444       # jmp to PLT[0]

<other PLT entries>
```

图 7.18 可执行文件 P2 的 PLT

原始代码见图 7.5 和图 7.6。

初始地，在程序被动态链接并开始执行后，过程 printf 和 addvec 被分别绑定到它们相应的 PLT 表目中的第一条指令上。比如，对 addvec 的调用有如下形式：

```
80485bb: e8 a4 fe ff ff call 8048464 <addvec>
```

当 addvec 第一次被调用时，控制传递到 PLT[2]的第一条指令，该指令通过 GOT[4]执行一个间接跳转。初始地，每个 GOT 表目包含相应的 PLT 表目中 pushl 表目的地址。所以，PLT 中的间接跳转仅仅是将控制转移回到 PLT[2]中的下一条指令。这条指令将 addvec 符号的 ID 压入栈中。最后一条指令跳转到 PLT[0]，从 GOT[1]中将另外一个标识信息的字压入栈中，然后通过 GOT[2]间接跳转到动态链接器中。动态链接器用两个栈表目来确定 addvec 的位置，用这个地址覆盖 GOT[4]，并把控制传递给 addvec。

下一次在程序中调用 addvec 时，控制像前面一样传递给 PLT[2]。不过这次通过 GOT[4]的间接跳转将控制传递给 addvec。从此刻起，惟一额外的开销就是对间接跳转的存储器引用。

7.13 处理目标文件的工具

在 Unix 系统中有大量可用的工具可以帮助你理解和处理目标文件。特别地，GNU binutils 包尤其有帮助，而且可以运行在每个 Unix 平台上。

- AR：创建静态库，插入、删除、列出和提取成员。
- STRINGS：列出一个目标文件中所有可打印的字符串。
- STRIP：从目标文件中删除符号表信息。
- NM：列出一个目标文件的符号表中定义的符号。
- SIZE：列出目标文件中节的名字和大小。
- READELF：显示一个目标文件的完整结构，包括 ELF 头中编码的所有信息。包含 SIZE 和 NM 的功能。
- OBJDUMP：所有二进制工具之母。能够显示一个目标文件中所有的信息。它最有用的功能是反汇编.text 节中的二进制指令。

Unix 系统为操作共享库还提供了 ldd 程序：

- ldd：列出一个可执行文件在运行时所需要的共享库。

7.14 小结

链接可以在编译时由静态编译器来完成，也可以在加载时和运行时由动态链接器来完成。链接器处理称为目标文件的二进制文件，它有三种不同的形式：可重定位的、可执行的和共享的。可重定位的目标文件由静态链接器组合成一个可执行的目标文件，它可以加载到存储器中并执行。共享目标文件（共享库）是在运行时由动态链接器链接和加载的，或者隐含地在调用程序被加载和开始执行时，或者根据需要在程序调用 dlopen 库的函数时。

链接器的两个主要任务是符号解析和重定位。符号解析将目标文件中的每个全局符号都绑定到一个惟一的定义，而重定位确定每个符号的最终存储器地址，并修改对那些目标的引用。

静态链接器是由像 GCC 这样的编译器调用的。它们将多个可重定位目标文件组合成一个单独的可执行目标文件。多个目标文件可以定义相同的符号，而链接器用来悄悄地解析这些多处定义的规则可能在用户程序中引入的微妙错误。

多个目标文件可以被连接到一个单独的静态库中。链接器用库来解析其他目标模块中的符号引用。许多链接器通过从左到右的顺序扫描来解析符号引用，这是另一个引起令人迷惑的链接时错误的来源。

加载器将可执行文件的内容映射到存储器，并运行这个程序。链接器还可能生成部分链接的可执行目标文件，这样的文件中有未解析的到定义在共享库中的程序和数据的引用。在加载时，加载器将部分链接的可执行文件映射到存储器，然后调用动态链接器，它通过加载共享库和重定位程序中的引用来完成链接任务。

被编译为位置无关代码的共享库可以加载到任何地方，也可以在运行时被多个进程共享。为了加载、链接和访问共享库的函数和数据，应用程序还可以在运行时使用动态链接器。

参考文献说明

在计算机系统文献中并没有很好地记录链接。因为链接是处在编译器、计算机体系结构和操作系统的交叉点上，它要求理解代码生成、机器语言编程、程序实例化和虚拟存储器。它恰好不落在某个通常的计算机系统专业中，因此这些领域的经典文献并没有很好地描述它。然而，Levine 的专著提供了有关这个主题的很好的一般性参考资料[47]。[35]描述了 ELF 和 DWARF 的原始规范（对.debug 和.line 节内容的规范说明）。

围绕二进制翻译（binary translation）的概念有一些有趣的研究和商业活动，二进制翻译包括目标文件内容的语法解析、分析和修改。二进制翻译有三个不同的目的[46]：在一个系统上模拟另一个系统，观察程序行为，或是执行不能在运行时执行的与系统相关的优化。一些商业产品，比如 VTune、Purify 和 BoundsChecker，用二进制翻译来为程序员提供对他们程序的详细的观察。

Atom 系统提出了一个灵活的机制，能为 Alpha 可执行目标文件和共享库提供任意的 C 函数。Atom 被用来创建无数种分析工具，包括跟踪过程调用、剖析指令计数和存储器引用模式、模拟存储器系统行为，以及隔离存储器引用错误。Etch[66]和 EEL[46]在不同的平台上提供了大致相似的功

能。Shade 系统利用二进制翻译实现指令剖析[15]。Dynamo[2]和 Dyninst[8]提供了一些机制，能在运行时为存储器中的可执行文件提供测试和优化。Smith 和他的同事们致力于研究程序剖析和优化的二进制翻译[91]。

家庭作业

7.6 ◆

考虑下面的 swap.c 函数，它计算自己被调用的次数：

```
1   extern int buf[];
2
3   Int *bufp0 = &buf[0];
4   static int *bufp1;
5
6   static void incr()
7   {
8     static int count=0;
9
10    count++;
11  }
12
13  void swap()
14  {
15       int temp;
16
17       incr();
18       bufp1 = &buf[1];
19       temp = *bufp0;
20       *bufp0 = *bufp1;
21       *bufp1 = temp;
22  }
```

对于每个 swap.o 中定义和引用的符号，如果它在模块 swap.o 的.symtab 节中有符号表表目，请指出。如果是这样，请指出定义该符号的模块（swap.o 或 main.o）、符号类型（本地、全局或外部）以及它在模块中所处的节（.text、.data 或.bss）。

符号	swap.o.symtab 表目？	符号类型	定义符号的模块	节
buf				
bufp0				
bufp1				
swap				
temp				
incr				
count				

7.7 ◆

不改变任何变量名字，修改 7.6.1 小节的 bar5.c，使得 foo5.c 输出 x 和 y 的正确值（也就是整数 15213 和 15212 的十六进制表示）。

7.8 ◆

在此题中，REF(x,i) --> DEF(x, k)表示链接器将任意对模块 i 中符号 x 的引用与模块 k 中符号 x 的定义相关联。在下面每个例子中，用这种符号来说明链接器是如何解析对在每个模块中有多个定义的引用的。如果出现链接时错误（规则 1），输出“ERROR”。如果链接器从定义中任意选择一个，那么输出“UNKNOWN”。

A.

```
/* Module 1 */                /* Module 2 */
int main()                    static int main=1;
{                             int p2()
}                             {
                              }
```

```
(a) REF(main.1) --> DEF(_____.___)
(b) REF(main.2) --> DEF(_____.___)
```

B.

```
/* Module 1 */                /* Module 2 */
int x;                        double x;
void main()                   int p2()
{                             {
}                             }
```

```
(a) REF(x.1) --> DEF(_____.__)
(b) REF(x.2) --> DEF(_____.__)
```

C.

```
/* Module 1 */                /* Module 2 */
int x=1;                      double x=1.0;
void main()                   int p2()
{                             {
}                             }
```

```
(a) REF(x.1) --> DEF(_____.___)
(b) REF(x.2) --> DEF(_____.___)
```

7.9 ◆

考虑下面的程序，它由两个目标模块组成：

```
1   /* foo6.c */
2   void p2(void);
3
4   int main()
```

```
5    {
6        p2();
7        return 0;
8    }
1    /* bar6.c */
2    #include <stdio.h>
3
4     char main;
5
6     void p2()
7     {
8         printf("0x%x\n", main);
9    }
```

当在 Linux 系统中编译和执行这个程序时，即使 p2 不初始化变量 main，它也能打印字符串“0x55\n”并正常终止。你能解释这一点吗？

7.10 ◆

a 和 b 表示当前路径中的目标模块或静态库，而 a→b 表示 a 依赖于 b，也就是说 a 引用了一个 b 定义的符号。对于下面的每个场景，给出使得静态链接器能够解析所有符号引用的最小的命令行（也就是，含有最少数量的目标文件和库参数的命令）。

A. p.o → libx.a → p.o.
B. p.o → libx.a → liby.a **and** liby.a → libx.a.
C. p.o → libx.a → liby.a → libz.a **and** liby.a → libx.a → libz.a.

7.11 ◆

图 7.12 中的段头表明数据段占用了存储器中 0x104 个字节。然而，只有开始的 0xe8 字节来自可执行文件的节。是什么引起了这种差异？

7.12 ◆◆

图 7.10 中的 swap 程序包含 5 个重定位的引用。对于每个重定位的引用，给出它在图 7.10 中的行号、它的运行时存储器地址和它的值。swap.o 模块中的原始代码和重定位表目如图 7.19 所示。

图7.10中的行号	地址	值

```
1    00000000 <swap>:
2      0: 55                    push    %ebp
3      1: 8b 15 00 00 00 00     mov     0x0,%edx          get *bufp0=&buf[0]
4                               3: R_386_32      bufp0    relocation entry
5      7: a1 04 00 00 00        mov     0x4,%eax          get buf[1]
```

```
6                                       8: R_386_32      buf       relocation entry
7     c: 89 e5                          mov     %esp,%ebp
8     e: c7 05 00 00 00 00 04           movl    $0x4,0x0           bufp1 = &buf[1];
9    15: 00 00 00
10                                      10: R_386_32     bufp1     relocation entry
11                                      14: R_386_32     buf       relocation entry
12   18: 89 ec                          mov %ebp,%esp
13   1a: 8b 0a                          mov     (%edx),%ecx        temp = buf[0];
14   1c: 89 02                          mov     %eax,(%edx)        buf[0]=buf[1];
15   1e: a1 00 00 00 00                 mov     0x0,%eax           get *bufp1=&buf[1]
16                                      1f: R_386_32     bufp1     relocation entry
17   23: 89 08                          mov     %ecx,(%eax)        buf[1]=temp;
18   25: 5d                             pop     %ebp
19   26: c3                             ret
```

图 7.19　练习题 7.12 的代码和重定位表目

7.13 ◆◆◆

考虑图 7.20 中的 C 代码和相应的可重定位目标模块。

A. 确定当模块被重定位时，链接器将修改.text 中的哪些指令。对与每条这样的指令，列出它的重定位表目中的信息：节偏移、重定位类型和符号名字。

B. 确定当模块被重定位时，链接器将修改.data 中的哪些数据目标。对于每条这样的指令，列出它的重定位表目中的信息：节偏移、重定位类型和符号名字。

可以随便使用诸如 OBJDUMP 之类的工具来帮助你解答这个题目。

```
1    extern int p3(void);
2    int x = 1;
3    int *xp = &x;
4
5    void p2(int y) {
6    }
7
8    void p1() {
9        p2(*xp + p3());
10   }
```

(a) C 代码

```
1    00000000 <p2>:
2       0: 55                        push    %ebp
3       1: 89 e5                     mov     %esp,%ebp
4       3: 89 ec                     mov     %ebp,%esp
```

```
5      5: 5d                       pop     %ebp
6      6: c3                       ret
7   00000008 <p1>:
8      8: 55                       push    %ebp
9      9: 89 e5                    mov     %esp,%ebp
10     b: 83 ec 08                 sub     $0x8,%esp
11     e: 83 c4 f4                 add     $0xfffffff4,%esp
12    11: e8 fc ff ff ff           call    12 <p1+0xa>
13    16: 89 c2                    mov     %eax,%edx
14    18: a1 00 00 00 00           mov     0x0,%eax
15    1d: 03 10                    add     (%eax),%edx
16    1f: 52                       push    %edx
17    20: e8 fc ff ff ff           call    21 <p1+0x19>
18    25: 89 ec                    mov     %ebp,%esp
19    27: 5d                       pop     %ebp
20    28: c3                       ret
```

(b) 可重定位目标文件的.text节

```
1   00000000 <x>:
2      0: 01 00 00 00
3   00000004 <xp>:
4      4: 00 00 00 00
```

(c) 可重定位目标文件的data节

图 7.20 练习题 7.13 的示例代码

7.14 ◆◆◆

考虑图 7.21 中的 C 代码和相应的可重定位目标模块。

A. 确定当模块被重定位时，链接器将修改.text 中的哪些指令。对于每条这样的指令，列出它的重定位表目中的信息：节偏移、重定位类型和符号名字。

B. 确定当模块被重定位时，链接器将修改.rodata 中的哪些数据。对于每条这样的指令，列出它的重定位表目中的信息：节偏移、重定位类型和符号名字。

可以随便使用诸如 OBJDUMP 之类的工具来帮助你解答这个题目。

```
1   int relo3(int val) {
2       switch (val) {
3       case 100:
4           return(val);
5       case 101:
6           return(val+1);
7       case 103: case 104:
```

```
8           return(val+3);
9       case 105:
10          return(val+5);
11      default:
12          return(val+6);
13      }
14  }
```

(a) C代码

```
1   00000000 <relo3>:
2      0: 55                        push   %ebp
3      1: 89 e5                     mov    %esp,%ebp
4      3: 8b 45 08                  mov    0x8(%ebp),%eax
5      6: 8d 50 9c                  lea    0xffffff9c(%eax),%edx
6      9: 83 fa 05                  cmp    $0x5,%edx
7      c: 77 17                     ja     25 <relo3+0x25>
8      e: ff 24 95 00 00 00 00      jmp    *0x0(,%edx,4)
9     15: 40                        inc    %eax
10    16: eb 10                     jmp    28 <relo3+0x28>
11    18: 83 c0 03                  add    $0x3,%eax
12    1b: eb 0b                     jmp    28 <relo3+0x28>
13    1d: 8d 76 00                  lea    0x0(%esi),%esi
14    20: 83 c0 05                  add    $0x5,%eax
15    23: eb 03                     jmp    28 <relo3+0x28>
16    25: 83 c0 06                  add    $0x6,%eax
17    28: 89 ec                     mov    %ebp,%esp
18    2a: 5d                        pop    %ebp
19    2b: c3 ret
```

(b) 可重定位目标文件的.text节

```
1   This is the jump table for the switch statement
2   0000 28000000 15000000 25000000 18000000  4 words at offsets 0x0,0x4, 0x8, and 0xc
3   0010 18000000 20000000                    2 words at offsets 0x10 and 0x14
```

(c) 可重定位目标文件的.rodata节

图 7.21 练习题 7.14 的示例代码

7.15 ◆◆◆

完成下面的任务将帮助你更熟悉处理目标文件的各种工具。

A. 在你的系统上，lib.c 和 libm.a 的版本中包含多少目标文件？

B. gcc -O2 产生的可执行代码与 gcc -O2 -g 产生的不同吗？

C. 在你的系统上，GCC 驱动程序使用的是什么共享库？

练习题答案

练习题 7.1 答案

这道练习题的目的是帮助你理解链接器符号和 C 变量及函数之间的关系。注意 C 的本地变量 temp 没有符号表表目。

符号	swap.o.symtab 表目?	符号类型	在哪个模块中定义	节
buf	是	extern	main.o	.data
bufp0	是	global	swap.o	.data
bufp1	是	global	swap.o	.bss
swap	是	global	swap.o	.text
temp	否	—	—	—

练习题 7.2 答案

这是一个简单的练习，检查你对 Unix 链接器解析定义在一个以上模块中的全局符号时所使用规则的理解。理解这些规则可以帮助你避免一些讨厌的编程错误。

A. 链接器选择定义在模块 1 中的强符号，而不是定义在模块 2 中的弱符号（规则 2）：

```
(a) REF(main.1) --> DEF(main.1)
(b) REF(main.2) --> DEF(main.1)
```

B. 这是一个错误，因为每个模块都定义了一个强符号 main（规则 1）。

C. 链接器选择定义在模块 2 中的强符号，而不是定义在模块 1 中的弱符号（规则 2）：

```
(a) REF(x.1) --> DEF(x.2)
(b) REF(x.2) --> DEF(x.2)
```

练习题 7.3 答案

在命令行中错误地放置静态库的位置是造成令许多程序员迷惑的链接器错误的常见原因。然而，一旦你理解了链接器是如何使用静态库来解析引用的，它就相当简单易懂了。这个小练习检查了你对这个概念的理解：

A. `gcc p.o libx.a`

B. `gcc p.o libx.a liby.a`

C. `gcc p.o libx.a liby.a libx.a`

练习题 7.4 答案

这道题涉及的是图 7.10 中的反汇编列表。在此，我们的目的是让你练习阅读反汇编列表，并检查你对 PC 相关寻址的理解。

A. 第 5 行被重定位引用的十六进制地址为 0x80483bb。

B. 第 5 行被重定位引用的十六进制值为 0x9。记住，反汇编列表给出了小端法字节顺序表示的

引用值。

C. 这里的关键观察点是无论链接器将.text 节定位在哪里，引用和 swap 函数间的距离总是一样的。因此，无论链接器将.text 节定位在何处，因为引用是一个 PC 相关地址，所以它的值都将是 0x9。

练习题 7.5 答案

对大多数程序员而言，C 程序实际是如何启动的是一个迷。这些问题检查了你对这个启动过程的理解。你可以参考图 7.14 中的 C 启动代码来回答这些问题：

A. 每个程序都需要一个 main 函数，因为 C 的启动代码对于每个 C 程序而言都是相同的，要跳转到一个叫做 main 的函数上。

B. 如果 main 以 return 语句终止，那么控制传递回启动程序，该程序通过调用_exit 再将控制返回给操作系统。如果用户省略了 return 语句，也会发生相同的情况。如果 main 是以调用 exit 终止的，那么 exit 将最终通过调用_exit 将控制返回给操作系统。在所有三种情况中，最终效果是相同的：当 main 完成时，控制会返回给操作系统。

CHAPTER 8

异常控制流

从给处理器加电开始，直到你断电为止，程序计数器假设一个序列的值

$$a_0, a_1, \cdots, a_{n-1}$$

其中，每个 a_k 是某个相应的指令 I_k 的地址。每次从 a_k 到 a_{k+1} 的过渡称为**控制转移**（control transfer）。这样的控制转移序列叫做处理器的**控制流**（flow of control 或 control flow）。

最简单的一种控制流是一个"平滑的"序列，其中每个 I_k 和 I_{k+1} 在存储器（memory）中都是相邻的。典型地，这种平滑流的突变，也就是 I_{k+1} 与 I_k 不相邻，是由诸如跳转、调用和返回这样一些熟悉的程序指令造成的。要想使得程序能够对由程序变量表示的内部的程序状态中的变化做出反应，这些指令是必要的机制。

但是系统也必须能够对系统状态的变化做出反应，这些系统状态不是被内部程序变量捕获的，而且也不一定要和程序的执行相关。比如，一个硬件定时器（或计时器）会定期产生信号，这个事件必须得到处理；包到达网络适配器后，必须存放在存储器中；程序向磁盘请求数据，然后休眠，直到被通知说数据已就绪；当子进程终止时，创造这些子进程的父进程必须得到通知。

现代系统通过使控制流发生突变来对这些情况做出反应。一般而言，我们把这些突变称为 ECF（exceptional control flow，*异常控制流*）。ECF 发生在计算机系统的各个层次。比如，在硬件层，硬件检测到的事件会触发控制突然转移到异常处理程序。在操作系统层，内核通过上下文转换将控制从一个用户进程转移到另一个用户进程。在应用层，一个进程可以发送一个信号到另一个进程，而接收者会将控制突然转移到它的一个信号处理程序。一个程序可以通过回避通常的栈规则，并执行到其他函数中任意位置的非本地跳转来对错误做出反应。

作为程序员，理解 ECF 对你们来说很重要，这有很多原因：

- *理解 ECF 将帮助你理解重要的系统概念*。ECF 是操作系统用来实现 I/O、进程和虚拟存储器的基本机制。在你可以真正理解这些重要概念之前，你必须理解 ECF。
- *理解 ECF 将帮助你理解应用程序是如何与操作系统交互的*。应用程序通过使用一个叫做陷阱（trap）或者系统调用（system call）的 ECF，向操作系统请求服务。比如，向磁盘写数据、从网络读取数据、创建一个新进程，以及终止当前进程，都是通过应用程序提交系统调用来实现的。理解基本的系统调用机制将帮助你理解这些服务是如何提供给应用的。
- *理解 ECF 将帮助你编写有趣的新应用程序*。操作系统为应用程序提供了强大的 ECF 机制，用来创建新进程、等待进程终止、通知其他进程系统中的异常事件，以及检测和响应这些事件。如果你理解这些 ECF 机制，那么你就能用它们来编写诸如 Unix shell 和 Web 服务器之类的有趣程序了。
- *理解 ECF 将帮助你理解软件异常如何工作*。像 C++和 Java 这样的语言通过 try、catch 以及 throw 语句来提供软件异常机制。软件异常允许程序进行非本地跳转（也就是，违反通常的调用/返回栈规则的跳转）来响应错误情况。非本地跳转是一种应用层 ECF，在 C 中是通过 setjmp 和 longjmp 函数提供的。理解这些低级函数将帮助你理解高级软件异常如何得以实现。

到目前为止，对系统的学习使你已经了解应用是如何与硬件交互的。这一章的重要性在于你将开始学习你的应用是如何与操作系统交互的。有趣的是，这些交互都是围绕着 ECF 的。我们描述存在于一个计算机系统中所有层次上的各种形式的 ECF。我们从异常开始，异常位于硬件和操作系统交界的部分。我们还会讨论系统调用，它们是为应用程序提供到操作系统的入口点的异常。然后，我们会提升抽象的层次，描述进程和信号，它们位于应用和操作系统的交界之处。最后，我们将讨

论非本地跳转，这是ECF的一种应用层形式。

8.1 异常

异常是一种形式的异常控制流，它一部分是由硬件实现的，一部分是由操作系统实现的。因为它们有一部分是由硬件实现的，所以具体细节将随系统的不同而有所不同。然而，对于每个系统而言，基本的思想都是相同的。在这一节中我们的目的是让你对异常和异常处理有一个一般性的了解，并且帮助消除现代计算机系统的一个经常令人感到迷惑的方面。

异常（exception）就是控制流中的突变，用来响应处理器状态中的某些变化。图 8.1 展示了基本的思想。

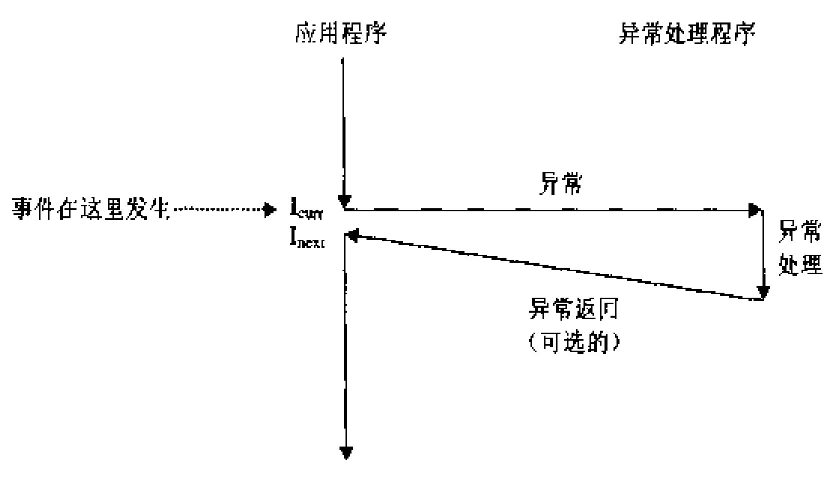


图 8.1 异常的剖析

处理器状态中的一个变化（事件）触发了从应用程序到一个异常处理程序的突发的控制转移（一个异常）。在异常处理程序完成处理后，它将控制返回给被中断的程序或者终止。

在此图中，当处理器状态中一个重要的变化发生时，处理器正在执行某个当前指令 I_{curr}。在处理器中，状态被编码为不同的位和信号。状态变化被称为*事件*（event），事件可能和当前指令的执行直接相关。比如，发生虚拟存储器缺页、算术溢出，或者一条指令试图除以零。另一方面，事件可能和当前指令的执行没有关系。比如，一个系统定时器产生信号或者一个I/O请求完成。

在任何情况中，当处理器检测到有事件发生时，它就会通过一张叫做*异常表*（exception table）的跳转表，进行一个间接过程调用（异常），到一个专门设计用来处理这类事件的操作系统子程序——*异常处理程序*（exception handler）。

当异常处理程序完成处理后，根据引起异常的事件的类型，会发生以下三种情况中的一种：

1. 处理程序将控制返回给当前指令 Icurr（当事件发生时正在执行的指令）。
2. 处理程序将控制返回给 Inext（如果没有发生异常将会执行的下一条指令）。
3. 处理程序终止被中断的程序。

8.1.2 节将讲述关于这些可能性的更多内容。

旁注：硬件与软件异常

C++和 Java 的程序员会注意到术语“异常”也用来描述由 C++和 Java 以 catch、throw 和 try 语

句的形式提供的应用级 ECF。如果我们想完全弄清楚，我们必须区别“硬件”和“软件”异常，但是这通常是不必要的，因为从上下文中就能够很清楚地知道是哪种含义。

8.1.1 异常处理

异常可能会难以理解，因为处理异常需要硬件和软件紧密合作。很容易搞混哪个部分执行哪个任务，让我们更详细地来看看硬件和软件的分工吧。

系统中可能的每种类型的异常都分配了一个惟一的非负整数的异常号（exception number）。这些号码中的某一些是由处理器的设计者分配的，其他号码是由操作系统内核（操作系统常驻存储器的部分）的设计者分配的。前者的示例包括被零除、缺页、存储器访问违例、断点以及算术溢出。后者的示例包括系统调用和来自外部 I/O 设备的信号。

在系统启动时（当计算机重启或者加电时），操作系统分配和初始化一张称为异常表的跳转表，使得表目 k 包含异常 k 的处理程序的地址。图 8.2 展示了一张异常表的格式。

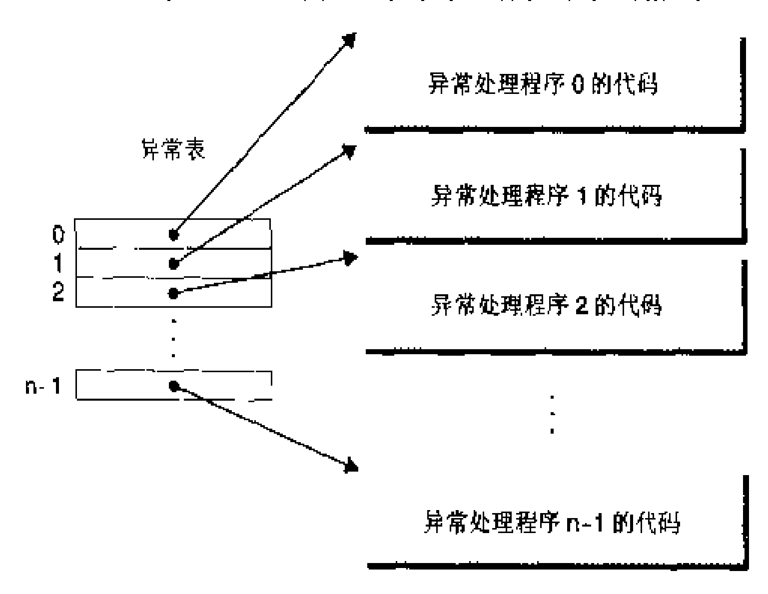


图 8.2 异常表

异常表是一张跳转表，其中表目 k 包含异常 k 的处理程序代码的地址。

在运行时（当系统在执行某个程序时），处理器检测到发生了一个事件，并且确定了相应的异常号 k。随后，处理器触发异常，方法是执行间接过程调用，通过异常表的表目 k，转到相应的处理程序。图 8.3 展示了处理器如何使用异常表来形成适当的异常处理程序的地址。异常号是到异常表中的索引，异常表的起始地址放在一个叫做*异常表基寄存器*（exception table base register）的特殊 CPU 寄存器里。

异常类似于过程调用，但是有一些重要的不同之处：

- 过程调用时，在跳转到处理程序之前，处理器将返回地址压到栈中。然而，根据异常的类型，返回地址要么是当前指令（当事件发生时正在执行的指令），要么是下一条指令（如果事件不发生，将会在当前指令后执行的指令）。
- 处理器也把一些额外的处理器状态压到栈里，在处理程序返回时，重新开始被中断的程序会需要这些状态。比如，一个 IA32 系统将包含当前条件码的 EFLAGS 寄存器和其他一些东西压入栈中。

- 如果控制从一个用户程序转移到内核，所有这些项目（item）都被压到内核栈中，而不是压到用户栈中。
- 异常处理程序运行在内核模式下（8.2.3 节），这意味着它们对所有的系统资源都有完全的访问权限。

一旦硬件触发了异常，剩下的工作就是由异常处理程序在软件中完成。在处理程序处理了事件之后，它通过执行一条特殊的"从中断返回"指令，可选地返回到被中断的程序，该指令将适当的状态弹回到处理器的控制和数据寄存器中，将状态恢复为用户模式（8.2.3 节）。如果异常中断的是一个用户程序，然后将控制返回给被中断的程序。

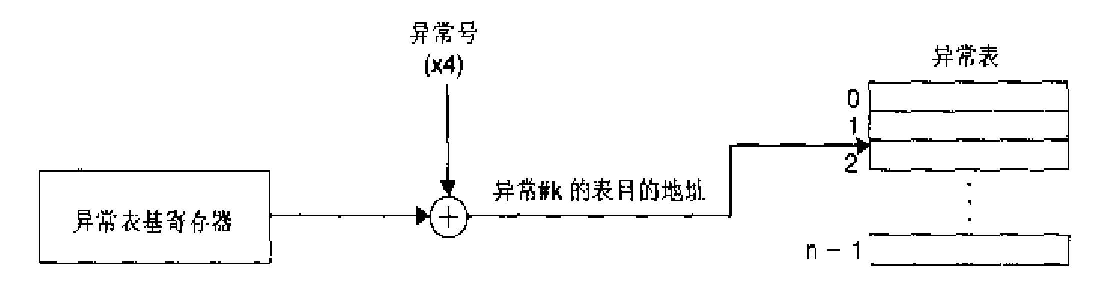


图 8.3 生成异常处理程序的地址

异常号是到异常表中的索引。

8.1.2 异常的类别

异常可以分为四类：中断（interrupt）、陷阱（trap）、故障（fault）和终止（abort）。图 8.4 中的表对这些类别的属性做了小结。

类别	原因	异步/同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

图 8.4 异常的类别

异步异常是由处理器外部的 I/O 设备中的事件产生的。同步异常是执行一条指令的直接产物。

中断

中断是异步发生的，是来自处理器外部的 I/O 设备的信号的结果。硬件中断不是由任何一条专门的指令造成的，从这个意义上来说它是异步的。硬件中断的异常处理程序常常被称为中断处理程序（interrupt handler）。

图 8.5 概述了一个中断的处理。I/O 设备，例如网络适配器、磁盘控制器和定时器芯片，通过向处理器芯片上的一个管脚发信号，并将异常号放到系统总线上，来触发中断，这个异常号标识了引起中断的设备。

在当前指令完成执行之前，处理器注意到中断管脚的电压变高了，就从系统总线读取异常号，然后调用适当的中断处理程序。当处理程序返回时，它就将控制返回给下一条指令（也就是，如果没有发生中断，在控制流中会在当前指令之后的那条指令）。结果是程序继续执行，就好像没有发生

过中断一样。

剩下的异常类型（陷阱、故障和终止）是同步发生的，是执行当前指令的结果。我们把这类指令叫做故障指令（faulting instruction）。

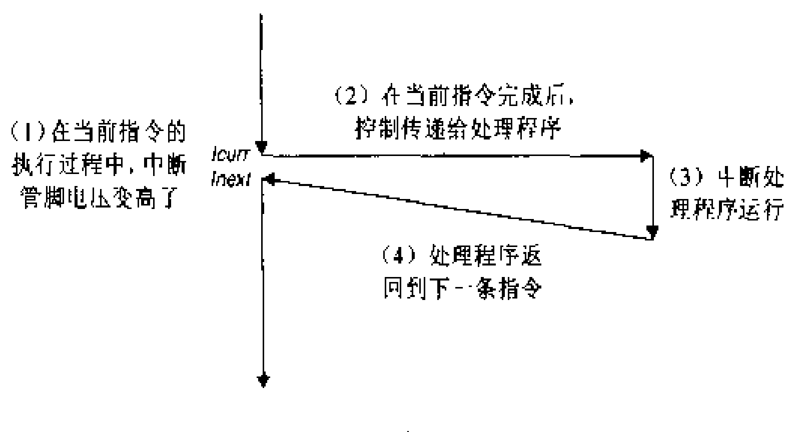


图 8.5 中断处理

中断处理程序将控制返回给应用程序控制流中的下一条指令。

陷阱

陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。

用户程序经常需要向内核请求服务，比如读一个文件（read）、创建一个新的进程（fork）、加载一个新的程序（execve），或者终止当前进程（exit）。为了允许对这些内核服务的受控的访问，处理器提供了一条特殊的“syscall n”指令，当用户程序想要请求服务 n 时，可以执行这条指令。执行 syscall 指令会导致一个到异常处理程序的陷阱，这个处理程序对参数解码，并调用适当的内核程序。图 8.6 概述了一个系统调用的处理。

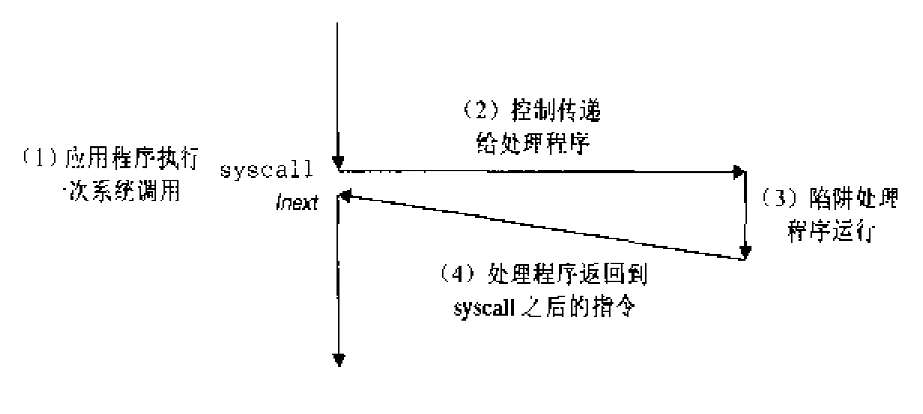


图 8.6 陷阱处理

陷阱处理程序将控制返回给应用程序控制流中的下一条指令。

从一个程序员的角度来看，系统调用和普通的函数调用是一样的。然而，它们的实现是非常不同的。普通的函数运行在用户模式（user mode）中，用户模式限制了函数可以执行的指令的类型，而且它们只能访问与调用函数相同的栈。系统调用运行在内核模式（kernel mode）中，内核模式允许系统调用执行指令，并访问定义在内核中的栈。8.2.3 节会更详细地讨论用户模式和内核模式。

故障

故障由错误情况引起，它可能被故障处理程序修正。当一个故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制返回到故障指令，从而重新执行它。否则，处理程序返回到内核中的 abort 例程，abort 例程会终止引起故障的应用程序。图 8.7 概述了一个故障的处理。

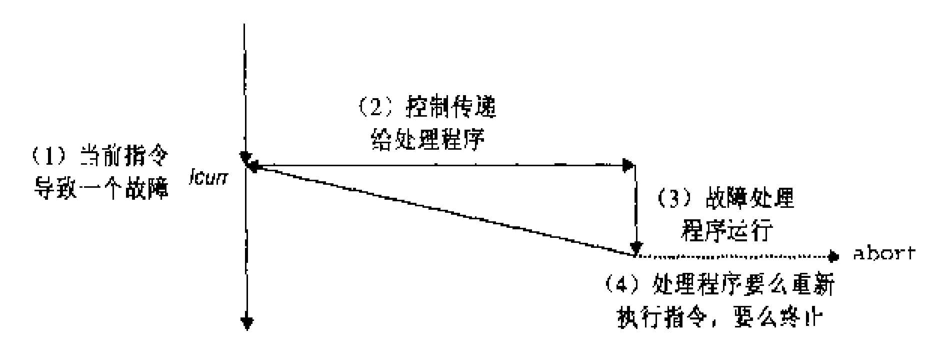


图 8.7 故障处理

根据故障是否能够被修复，故障处理程序要么重新执行故障指令，要么终止。

故障的一个经典示例是缺页异常，当指令引用一个虚拟地址，而与该地址相对应的物理页面不在存储器中，因此必须从磁盘中取出时，就会发生这种故障。就像我们将在第 10 章中看到的那样，一个页面就是虚拟存储器的一个连续的块（典型的是 4KB）。缺页处理程序从磁盘加载适当的页面，然后将控制返回给引起故障的指令。当指令再次执行时，相应的物理页面已经驻留在存储器中了，指令就可以没有故障地运行完成了。

终止

终止是不可恢复的致命错误造成的结果——典型的是一些硬件错误，比如 DRAM 或者 SRAM 位被损坏时发生的奇偶错误。终止处理程序从不将控制返回给应用程序。如图 8.8 所示，处理程序将控制返回给一个 abort 例程，该例程会终止这个应用程序。

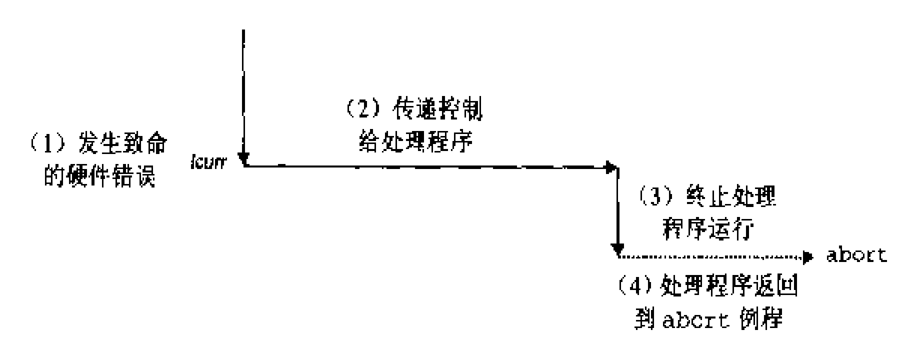


图 8.8 终止处理

终止处理程序将控制传递给一个内核 abort 例程，该例程会终止这个应用程序。

8.1.3 Intel 处理器中的异常

为了使描述更具体，让我们来看看为 Intel 系统定义的一些异常。一个 Pentium 系统可以有高达 256 种不同的异常类型。范围 0～31 的号码对应的是 Pentium 体系结构定义的异常，因此对任何 Pentium 类的系统都是一样的。范围 32～255 的号码对应的是操作系统定义的中断和陷阱。图 8.9 展

示了一些示例。

当应用试图除以零时，或者当一个除法指令的结果对于目标操作数来说太大了时，就会发生除法错误（异常0）。Unix不会试图从除法错误中恢复，而是选择终止程序。Unix shell典型地会把除法错误报告为“浮点异常（Floating exception）”。

异常号	描述	异常类别
0	除法错误	故障
13	一般保护故障	故障
14	缺页	故障
18	机器检查	终止
32～127	操作系统定义的异常	中断或陷阱
128（0x80）	系统调用	陷阱
129～255	操作系统定义的异常	中断或陷阱

图8.9 Pentium系统中的异常示例

许多原因都会导致不为人知的一般保护故障（异常13），通常是因为一个程序引用了一个未定义的虚拟存储器区域，或者因为程序试图写一个只读的文本段。Unix不会尝试恢复这类故障。Unix shell典型地将这种一般保护故障报告为“段故障（Segmentation fault）”。

缺页（异常14）是会重新执行故障指令的异常的一个示例。处理程序将磁盘上物理存储器相应的页面映射到虚拟存储器的一个页面，然后重新开始这条故障指令。我们将在第10章中看到这是如何工作的细节。

机器检查（异常18）是在故障指令执行中检测到致命的硬件错误时发生的。机器检查处理程序从不返回控制给应用程序。

在IA32系统上，系统调用是通过一条称为INT n的陷阱指令来提供的，其中n可能是异常表中256个表目中任何一个的索引。在历史上，系统调用是通过异常128（0x80）提供的。

旁注：关于术语的注释

各种异常类型的术语是根据系统的不同而有所不同的。处理器微体系结构规范通常会区分异步的“中断”和同步的“异常”，但是并不提供描述这些非常相似的概念的umbrella术语。为了避免不断地提到“异常和中断”以及“异常或者中断”，我们用单词“异常”作为通用的术语，而且只有在必要时才区别异步异常（中断）和同步异常（陷阱、故障和终止）。正如我们提到过的，对于每个系统而言，基本的概念都是相同的，但是你应该意识到一些制造厂商的手册会用“异常”仅仅表示同步事件引起的控制流中的那些改变。

8.2 进程

异常提供基本的构造块，它允许操作系统提供进程（process）的概念，进程是计算机科学中最深刻最成功的概念之一。

当我们在一个现代系统上运行一个程序时，我们会得到一个假象，就好像我们的程序是系统中当前运行的惟一程序。我们的程序好像是独占地使用处理器和存储器。处理器就好像是无间断的一

条接一条地执行我们程序中的指令。最后，我们程序中的代码和数据显得好像是系统存储器中惟一的对象。这些假象都是通过进程的概念提供给我们的。

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都是运行在某个进程的上下文（context）中的。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在存储器中的程序的代码和数据、它的栈、它的通用目的寄存器的内容、它的程序计数器、环境变量以及打开文件描述符的集合。

每次用户通过向 shell 输入一个可执行目标文件的名字，运行一个程序时，shell 会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序还能够创建新进程，且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

关于操作系统如何实现进程的细节的讨论超出了我们的范围。取而代之，我们将关注进程提供给应用程序的关键抽象：

- 一个独立的逻辑控制流，它提供一个假象，使我们觉得我们的程序独占地使用处理器。
- 一个私有的地址空间，它提供一个假象，使我们觉得我们的程序独占地使用存储器系统。

让我们更深入地看看这些抽象。

8.2.1 逻辑控制流

典型地，即使在系统中有许多其他程序在运行，进程也可以向每个程序提供一种假象，好像它在独占地使用处理器。如果我们想用调试器单步执行我们的程序，我们会看到一系列的 PC（程序计数器）的值，这些值惟一地对应于包含在我们程序的可执行目标文件中的指令或是包含在运行时动态链接到我们程序的共享对象中的指令。这个 PC 值的序列叫做逻辑控制流。

考虑一个运行着三个进程的系统，如图 8.10 所示。处理器的一个物理控制流被分成了三个逻辑流，每个进程一个。每一个竖直方向上的列表示一个进程的逻辑流的一部分。在这个例子中，进程 A 运行了一会儿，然后是 B 开始运行到完成。然后，C 运行了一会儿，A 接着运行直到完成。最后，C 可以运行到结束了。

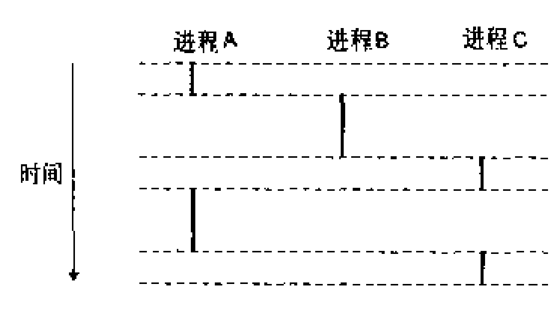


图 8.10 逻辑控制流

进程为每个程序提供了一种假象，好像程序在独占地使用处理器。每一竖直方向上的列表示一个进程的逻辑控制流的一部分。

图 8.10 的关键点在于进程是轮流使用处理器的。每个进程执行它的流的一部分，然后被抢占(preempted)（暂时挂起），与此同时其他进程开始执行。对于一个运行在这些进程之一的上下文中的程序，它看上去就像是在独占地使用处理器。惟一的反面例证是如果我们精确地测量每条指令使用的时间（参见第 9 章），我们将发现在我们程序中一些指令的执行之间，CPU 好像会周期性地停顿（stall）。然而，每次处理器停顿，它随后继续执行我们的程序，并不改变程序存储器位置或寄存器的内容。

一般而言，和不同进程相关的逻辑流并不影响任何其他进程的状态，从这个意义上说，每个逻

辑流都是与其他逻辑流相独立的。当进程使用进程间通信（IPC）机制，比如管道、套接口、共享存储器和信号量，显式地与其他进程交互时，这条规则的惟一例外就会发生。

任何逻辑流在时间上和另外的逻辑流重叠的进程被称为*并发进程*（concurrent process），而这两个进程就被称为并发运行。比如，图 8.10 中，进程 A 和 B 就是并发运行的，A 和 C 也是。另一方面，B 和 C 并不是并发运行的，因为 B 的最后一条指令是在 C 的第一条指令之前执行的。

进程和其他进程轮换运行的概念称为*多任务*（multitasking）。一个进程执行它的控制流的一部分的每一时间段叫做*时间片*（time slice）。因此，多任务也叫做*时间分片*（time slicing）。

8.2.2 私有地址空间

进程也为每个程序提供一种假象，就好像它正在独占地使用系统地址空间。在一台 n 位地址的机器上，地址空间是 2^n 个可能地址的集合，0，1，…，2^n-1。一个进程为每个程序提供它自己的*私有地址空间*。一般而言，和这个空间中某个地址相关联的那个存储器字节是不能被其他进程读或者写的，从这个意义上说，这个地址空间是私有的。

尽管和每个私有地址空间相关联的存储器的内容一般是不同的，但是每个这样的空间都有相同的结构。比如，图 8.11 展示了一个 Linux 进程的地址空间的结构。地址空间底部的四分之三是预留给用户程序的，包括通常的文本、数据、堆和栈段。地址空间顶部的四分之一是预留给内核的。地址空间的这个部分包含内核在代表进程执行指令时（比如，当应用程序执行一个系统调用时）使用的代码、数据和栈。

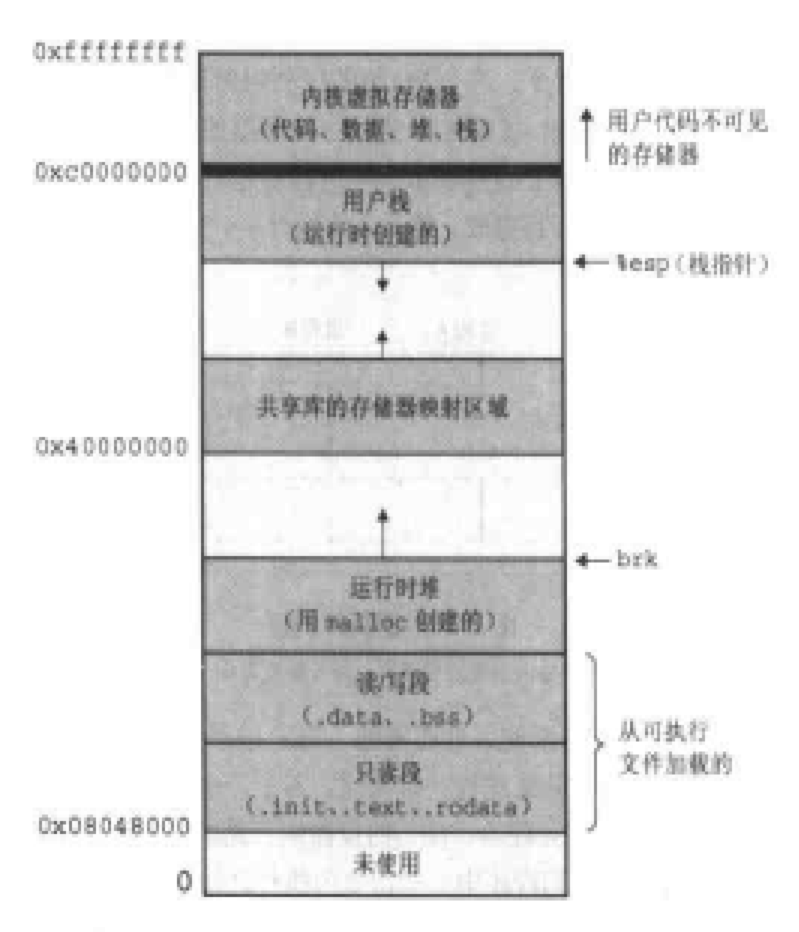


图 8.11 进程地址空间

8.2.3 用户模式和内核模式

为了使操作系统内核提供一个无懈可击的进程抽象，处理器必须提供一种机制，限制一个应用可以执行的指令以及它可以访问的地址空间范围。

典型地，处理器是用某个控制寄存器中的一个*方式位*（mode bit）来提供这种功能的，该寄存器描述了进程当前享有的权力。当方式位设置了时，进程就运行在*内核模式*中（有时叫做*超级用户模式*）。一个运行在内核模式的进程可以执行指令集中的任何指令，并且可以访问系统中任何存储器位置。

方式位没有设置时，进程就运行在*用户模式*中。用户模式中的进程不允许执行特权指令（privileged instruction），比如停止处理器、改变方式位的值或者发起一个I/O操作，也不允许用户模式中的进程直接引用地址空间中内核区内的代码和数据。任何这样的尝试都会导致致命的保护故障。用户程序必须通过系统调用接口间接地访问内核代码和数据。

一个运行应用程序代码的进程初始时是在用户模式中的。进程从用户模式变为内核模式的惟一方法是通过诸如中断、故障或者陷入系统调用（trapping system call）这样的异常。当异常发生时，控制传递到异常处理程序，处理器将模式从用户模式变为内核模式。处理程序运行在内核模式中，当它返回到应用代码时，处理器就把模式从内核模式改回到用户模式。

Linux和Solaris提供了一种聪明的机制，叫做/proc文件系统，它允许用户模式进程访问内核数据结构的内容。/proc文件系统将许多内核数据结构的内容输出为一种用户程序可以读的ASCII文件的层次结构。比如，你可以使用Linux /proc文件系统找出一般的系统属性，比如CPU类型（/proc/cpuinfo），或者某个进程使用的存储器段（/proc/<process id>/maps）。

8.2.4 上下文切换

操作系统内核利用一种称为*上下文切换*（context switch）的较高级形式的异常控制流来实现多任务。上下文切换机制是建立在我们在8.1节中已经讨论过的那些较低层异常机制之上的。

内核为每个进程维持一个*上下文*（context）。上下文就是内核重新启动一个被抢占进程所需的状态。它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描绘地址空间的*页表*（page table）、包含有关当前进程信息的*进程表*（process table），以及包含进程已打开文件的信息的*文件表*（file table）。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占的进程。这种决定就叫做*调度*（scheduling），是由内核中称为*调度器*（scheduler）的代码处理的。当内核选择一个新的进程运行时，我们就说内核调度了这个进程。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，*上下文切换*可以：①保存当前进程的上下文；②恢复某个先前被抢占进程所保存的上下文；③将控制传递给这个新恢复的进程。

当内核代表用户执行系统调用时，可以发生上下文切换。如果系统调用因为等待某个事件发生而阻塞，那么内核可以让当前进程休眠，切换到另一个进程。比如，如果一个read系统调用请求一个磁盘访问，内核可以选择执行上下文切换，运行另外一个进程，而不是等待数据从磁盘到达。另一个示例是sleep系统调用，它显式地请求让调用进程休眠。一般而言，即使系统调用没有阻塞，内核也可以决定执行上下文切换，而不是将控制返回给调用进程。

中断也可能引发上下文切换。比如，所有的系统都有某种产生周期性定时器中断的机制，典型的为每1毫秒或每10毫秒。每次发生定时器中断时，内核就能判定当前进程已经运行了足够长的时间了，并切换到一个新的进程。

图8.12展示了一对进程A和B之间上下文切换的示例。在这个例子中，初始地，进程A运行在用户模式中，直到它通过执行read系统调用陷入到内核。内核中的陷阱处理程序请求来自磁盘控制器的DMA传输，并在磁盘控制器完成从磁盘到存储器的数据传输后，要求磁盘中断处理器。

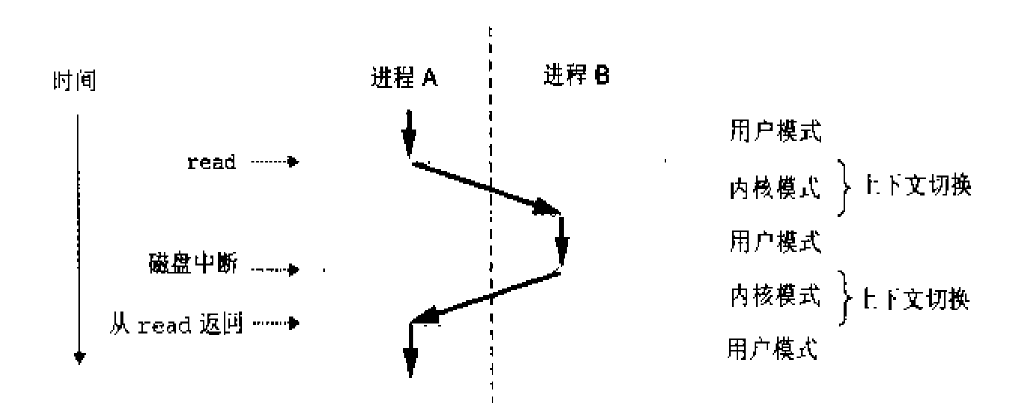


图8.12 进程上下文切换的剖析

磁盘取数据要用一段相对较长的时间（数量级为十几毫秒），所以内核执行从进程A到进程B的上下文切换，而不是在这个间歇时间内等待，什么都不做。注意在切换之前，内核正代表进程A在用户模式下执行指令。在切换的第一步中，内核代表进程A在内核模式下执行指令。然后在某一时刻，它开始代表进程B（仍然是内核模式下）执行指令。在切换完成之后，内核代表进程B在用户模式下执行指令。

随后，进程B在用户模式下运行一会儿，直到磁盘发出一个中断信号，表示数据已经从磁盘传送到了存储器。内核判定进程B已经运行了足够长的时间了，就执行一个从进程B到进程A的上下文切换，将控制返回给进程A中紧随在read系统调用之后的那条指令。进程A继续运行，直到下一次异常发生，依此类推。

旁注：高速缓存污染（pollution）和异常控制流

一般而言，硬件高速缓存存储器不能和诸如中断和上下文切换这样的异常控制流很好地交互。如果当前进程被一个“中断”暂时中断，那么对于中断处理程序来说高速缓存是冷的（cold）[1]。如果处理程序从主存中访问了足够多的表目，那么当被中断的进程继续时，高速缓存对它来说也是冷的了。在这种情况中，我们就说（中断）处理程序污染（pollute）了高速缓存。使用上下文切换也会发生类似的现象。当一个进程在上下文切换后继续执行时，高速缓存对于应用程序而言也是冷的，必须再次热身。

1 “高速缓存是冷的”意思是程序所需要的数据都不在高速缓存中。——译者

8.3 系统调用和错误处理

Unix 系统提供了大量的系统调用，当应用程序想向内核请求服务时，比如读取一个文件，或者创建一个新的进程，都可以使用这些系统调用。例如，Linux 提供了大约 160 个系统调用。输入"man syscalls"，你将得到完整的列表。

C 程序通过使用"man 2 intro"里描述的_syscall 宏，可以直接调用任何"系统调用"。然而，通常直接调用"系统调用"既不必要又不值得。标准 C 库提供了一组针对最常用系统调用的方便的包装（wrapper）函数。包装函数将参数打好包，通过适当的系统调用陷入内核，然后将系统调用的返回状态传递给调用程序。在我们下面章节的讨论中，我们把系统调用和它们相关的包装函数可互换地称为系统级函数。

当 Unix 系统级函数遇到错误时，它们典型地会返回-1，并设置全局整数变量 errno 来表示什么出错了。程序员应该总是检查这些错误，但是不幸的是，许多人都忽略了错误检查，因为它使代码变得臃肿，而且难以读懂。比如，下面是我们调用 Unix fork 函数时如何检查错误的：

```
1  if ((pid = fork()) < 0) {
2      fprintf(stderr, "fork error: %s\n", strerror(errno));
3      exit(0);
4  }
```

strerror 函数返回一个文本串，描述了和某个 errno 值相关联的错误。通过定义下面的错误报告函数（error-reporting function），我们能够在某种程度上简化这个代码：

```
1  void unix_error(char *msg) /* unix-style error */
2  {
3      fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4      exit(0);
5  }
```

给定这个函数，我们对 fork 的调用从 4 行简化到了 2 行：

```
1  if ((pid = fork()) < 0)
2      unix_error("fork error");
```

通过使用错误处理包装（error-handling wrapper）函数，我们可以更进一步地简化我们的代码。对于一个给定的基本函数 foo，我们定义一个具有相同参数的包装函数 Foo，但是第一个字母大写了。包装函数调用基本函数来检查错误，如果有任何问题就终止。比如，下面是 fork 函数的错误处理包装函数：

```
1  pid_t Fork(void)
2  {
3      pid_t pid;
4
5      if ((pid = fork()) < 0)
6          unix_error("Fork error");
7      return pid;
8  }
```

给定这个包装函数，我们对 fork 的调用就缩减为 1 行：

```
1  pid = Fork();
```

我们将在本书剩余的部分中都使用错误处理包装函数。它们允许我们保持示例代码的简洁，而又不会给你错误的假象，认为允许忽略错误检查。注意，当我们在本书中谈到系统级函数时，我们总是用它们的小写字母来表示，而不是它们大写的包装函数名来表示。

关于 Unix 错误处理以及本书中使用的错误处理包装函数的讨论，请参考附录 B。包装函数定义在一个叫做 csapp.c 的文件中，它们的原型定义在一个叫做 csapp.h 的头文件中。为了便于你引用，附录 B 提供了这些文件的源代码。

8.4 进程控制

Unix 提供了大量从 C 程序中操作进程的系统调用。这一节将描述这些重要的函数，并举例说明如何使用它们。

8.4.1 获取进程 ID

每个进程都有一个惟一的正数（非零）进程 ID（PID）。getpid 函数返回调用进程的 PID。getppid 函数返回它的父进程的 PID（也就是，创建调用进程的进程）。

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

返回：调用者或其父进程的 PID。

getpid 和 getppid 函数返回一个类型为 pid_t 的整数值，在 Linux 系统上的 types.h 中它被定义为 int。

8.4.2 创建和终止进程

从程序员的角度，我们可以认为进程总是处于下面三种状态之一：

- 运行。进程要么在 CPU 上执行，要么在等待被执行且最终会被调度。
- 暂停。进程的执行被挂起（suspended），且不会被调度。当收到 SIGSTOP、SIGTSTP、SIDTTIN 或者 SIGTTOU 信号时，进程就暂停，并且保持暂停直到它收到一个 SIGCONT 信号，在这个时刻，进程再次开始运行。（信号是一种软件中断的形式，将在 8.5 节中给予描述。）
- 终止。进程永远地停止了。进程会因为三种原因终止：收到一个信号，该信号的默认行为是终止进程；从主程序返回；调用 exit 函数。

```
#include <stdlib.h>

void exit(int status);
```

该函数无返回值。

exit 函数以 status 退出状态来终止进程（另一种设置退出状态的方法是从主程序中返回一个整数值）。

父进程通过调用 fork 函数创建一个新的运行子进程：

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

返回：子进程返回0，父进程返回子进程的PID，若出错则为-1。

新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份拷贝，包括文本、数据和bss段、堆以及用户栈。子进程还获得与父进程任何打开文件描述符相同的拷贝，这就意味着当父进程调用fork时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的PID。

fork函数是有趣的（也常常令人迷惑），因为它只被调用一次，却会返回两次：一次是在调用进程（父进程）中，一次是在新创建的子进程中。在父进程中，fork返回子进程的PID。在子进程中，fork返回零。因为子进程的PID总是非零的，返回值就提供一个明确的方法来分辨程序是在父进程还是在子进程中执行的。

图8.13展示了一个使用fork创建子进程的父进程的示例。当fork调用在第8行返回时，在父进程和子进程中x都有值1。子进程在第10行增加并输出它的x的拷贝。相似地，父进程在第15行减少和输出它的x的拷贝。

code/ecf/fork.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6       int x = 1;
7
8       pid = Fork();
9       if (pid == 0) { /* child */
10          printf("child : x=%d\n", ++x);
11          exit(0);
12      }
13
14      /* parent */
15      printf("parent: x=%d\n", --x);
16      exit(0);
17  }
```

code/ecf/fork.c

图8.13 使用fork创建一个新进程

当我们在Unix系统上运行这个程序时，我们得到下面的结果：

```
unix> ./fork
parent: x=0
child : x=2
```

这个简单的例子有一些微妙的方面：

- **调用一次，返回两次。** fork 函数被父进程调用一次，但是却返回两次——一次是返回到父进程，一次是返回到新创建的子进程。对于只创建一个子进程的程序来说，这还是相当简单的。但是含有多个 fork 实例的程序可能就会令人迷惑，需要仔细地推敲了。
- **并发执行。** 父进程和子进程是并发运行的独立进程。内核能够以任意方式交替执行它们逻辑控制流中的指令。当我们在系统上运行这个程序时，父进程先完成它的 printf 语句，然后是子进程。然而，在另一个系统上可能正好相反。一般而言，作为程序员，我们无法对不同进程中的指令交替执行做任何假设。
- **相同的但是独立的地址空间。** 如果我们能够在 fork 函数在父进程和子进程中返回后立即终止这两个进程，我们会看到每个进程的地址空间都是相同的。每个进程有相同的用户栈、相同的本地变量值、相同的堆、相同的全局变量值，以及相同的代码。因此，在我们的示例程序中，当 fork 函数在第 8 行返回时，本地变量 x 在父进程和子进程中都为 1。然而，因为父进程和子进程是独立的进程，它们每个都有自己的私有地址空间。后面，父进程和子进程对 x 所做的任何改变都是独立的，不会反映在另一个进程的存储器中。这就是为什么当父进程和子进程调用它们各自的 printf 函数时，它们中的变量 x 会有不同的值。
- **共享文件。** 当我们运行示例程序时，我们注意到父进程和子进程都把它们的输出显示在屏幕上。原因是子进程继承了父进程所有的打开文件。当父进程调用 fork 时，stdout 文件是被打开的，并指向屏幕。子进程继承了这个文件，因此它的输出也是指向屏幕的。

如果你是第一次学习 fork 函数，画进程图通常会有所帮助，其中每个水平的箭头对应于从左到右执行指令的进程，而每个垂直的箭头对应于 fork 函数的执行。

例如，图 8.14（a）中的程序将产生多少输出行呢？图 8.14（b）给出了相应的进程图。当父进程执行程序中第一个（也是惟一一个）fork 函数时，它会创建一个子进程。每个进程都调用一次 printf，所以程序打印两个输出行。

现在如果我们如图 8.14（c）所示的那样调用 fork 两次，会怎样呢？就像我们在图 8.14（d）中看到的那样，父进程调用 fork 创建一个子进程，然后父进程和子进程都调用 fork，这就导致了两个更多的进程。因此，就有了 4 个进程，每个都调用 printf，所以程序就产生了 4 个输出行。

继续沿这个思路想下去，如果我们要调用 fork 三次，如图 8.14（e）所示，又会发生什么呢？就像我们从图 8.14（f）中的进程图中看到的那样，一共会有 8 个进程。每个进程调用 printf，所以程序就产生了 8 个输出行。

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```

（a）调用 fork 一次

hello
hello
fork

（b）打印两个输出行

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

(c) 调用 fork 两次

(d) 打印四个输出行

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```

(e) 调用 fork 三次

(f) 打印八个输出行

图 8.14 fork 示例程序

练习题 8.1

考虑下面的程序：

code/ecf/forkprob0.c

```
1    #include "csapp.h"
2
3    int main()
4    {
5        int x = 1;
6
7        if (Fork() == 0)
8            printf("printf1: x=%d\n", ++x);
9        printf("printf2: x=%d\n", --x);
10       exit(0);
11   }
```

code/ecf/forkprob0.c

A. 子进程的输出是什么？

B. 父进程的输出是什么？

练习题 8.2

下面的程序会打印多少个“hello”输出行？

code/ecf/forkprob1.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int i;
6
7       for (i = 0; i < 2; i++)
8           Fork();
9       printf("hello!\n");
10      exit(0);
11  }
```

code/ecf/forkprob1.c

练习题 8.3

下面的程序会打印多少个“hello”输出行？

code/ecf/forkprob4.c

```
1   #include "csapp.h"
2
3   void doit()
4   {
5       Fork();
6       Fork();
7       printf("hello\n");
8       return;
9   }
10
11  int main()
12  {
13      doit();
14      printf("hello\n");
15      exit(0);
16  }
```

code/ecf/forkprob4.c

8.4.3 回收子进程

当一个进程由于某种原因终止时，内核并不是立即把它从系统中清除。取而代之的是，进程被保持在一种终止状态中，直到被它的父进程回收（reaped）。当父进程回收已终止的子进程时，内核

将子进程的退出状态传递给父进程，然后抛弃已终止的进程，从此时开始，该进程就不存在了。一个终止了但还未被回收的进程称为僵死进程（zombie）。

旁注：为什么已终止的子进程称为僵死进程？

在民间传说中，僵尸是活着的尸体，一种半生半死的实体。僵死进程已经终止了，而内核仍保留着它的某些状态直到父进程回收它为止，从这个意义上说它们是类似的。

如果父进程没有回收它的僵死子进程就终止了，那么内核就会安排 init 进程来回收它们。init 进程的 PID 为 1，并且是在系统初始化时由内核创建的。长时间运行的程序，比如 shell 或者服务器，总是应该回收它们的僵死子进程。即使僵死子进程没有运行，它们仍然消耗系统的存储器资源。

一个进程可以通过调用 waitpid 函数来等待它的子进程终止或者暂停：

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

返回：如果成功，则为子进程的 PID，如果 WNOHANG，则为 0，如果出错则为-1。

waitpid 函数有点复杂。默认地（当 options=0 时），waitpid 挂起调用进程的执行，直到它的等待集合中的一个子进程终止。如果等待集合中的一个进程在刚调用的时刻就已经终止了，那么 waitpid 就立即返回。在这两种情况中，waitpid 返回导致 waitpid 返回的终止子进程的 PID，并且将这个已终止的子进程从系统中去除。

判定等待集合的成员

等待集合的成员是由参数 pid 来确定的：

- 如果 pid>0，那么等待集合就是一个单独的子进程，它的进程 ID 等于 pid。
- 如果 pid=-1，那么等待集合就是由父进程所有的子进程组成的。

旁注：在进程集合上的等待

waitpid 函数还支持其他类型的等待集合，包括 Unix 进程组，对此我们将不做讨论。

修改默认行为

可以通过用常量 WNOHANG 和 WUNTRACED 的不同组合来设置 options，修改默认行为：

- WNOHANG：如果没有等待集合中的任何子进程终止，那么就立即返回（返回值为 0）。
- WUNTRACED：挂起调用进程的执行，直到等待集合中的一个进程变成终止的或者被暂停。返回的 PID 为导致返回的终止或暂停子进程的 PID。
- WNOHANG|WUNTRACED：立即返回，如果没有等待集合中的任何子进程停止或终止，那么返回值为 0，或者返回值等于那个被停止或者终止子进程的 PID。

检查已回收子进程的退出状态

如果 status 参数是非空的，那么 waitpid 就会编码关于导致返回的子进程的状态信息到 status 参数。wait.h 包含文件定义了解释 status 参数的几个宏：

- WIFEXITED(status)：如果子进程正常终止就返回真，也就是通过调用 exit 或者一个返回

(return)。

- WEXITSTATUS(status)：返回一个正常终止的子进程的退出状态。只有在 WIFEXITED 返回为真时，才会定义这个状态。
- WIFSIGNALED(status)：如果是因为一个未被捕获的信号造成了子进程的终止，那么就返回真（将在 8.5 节中解释说明信号）。
- WTERMSIG(status)：返回引起子进程终止的信号的数量。只有在 WIFSIGNALED(status)返回真时，才定义这个状态。
- WIFSTOPPED(status)：如果引起返回的子进程当前是暂停的，那么就返回真。
- WSTOPSIG(status)：返回引起子进程暂停的信号的数量。只有在 WIFSTOPPED（status）返回真时，才定义这个状态。

错误条件

如果调用进程没有子进程，那么 waitpid 返回-1，并且设置 errno 为 ECHILD。如果 waitpid 函数被一个信号中断，那么它返回-1，并设置 errno 为 EINTR。

旁注：和 Unix 函数相关的常量

像 WNOHANG 和 WUNTRACED 这样的常量是由系统头文件定义的。例如，WNOHANG 和 WUNTRACED 是由 wait.h 头文件（间接）定义的：

```
/* Bits in the third argument to 'waitpid'. */
#define WNOHANG      1 /* Don't block waiting. */
#define WUNTRACED    2 /* Report status of stopped children. */
```

为了使用这些常量，你必须在你的代码中包含 wait.h 头文件：

```
#include <sys/wait.h>
```

每个 Unix 函数的 man 页列出了无论何时你在代码中使用那个函数都要包含的头文件。同时，为了检查诸如 ECHILD 和 EINTR 之类的返回代码，你必须包含 errno.h。为了简化我们的代码示例，我们包含了一个称为 csapp.h 的头文件，它包括了本书中使用的所有函数的头文件。附录 B 中列出了 csapp.h 头文件。

示例

图 8.15 展示了一个创建 N 个子进程的程序，使用 waitpid 等待它们终止，然后查看每个终止子进程的退出状态。

当我们在 Unix 系统上运行这个程序时，它会产生如下输出：

```
unix ./waitpid1
child 22966 terminated normally with exit status = 100
child 22967 terminated normally with exit status = 101
```

code/ecf/waitpid1.c

```
1   #include "csapp.h"
2   #define N 2
3
4   int main()
```

```
5   {
6       int status, i;
7       pid_t pid;
8
9       for (i = 0; i < N; i++)
10          if ((pid = Fork()) == 0) /* child */
11              exit(100+i);
12
13      /* parent waits for all of its children to terminate */
14      while ((pid = waitpid(-1, &status, 0)) > 0) {
15          if (WIFEXITED(status))
16              printf("child %d terminated normally with exit status=%d\n",
17                     pid, WEXITSTATUS(status));
18          else
19              printf("child %d terminated abnormally\n", pid);
20      }
21      if (errno != ECHILD)
22          unix_error("waitpid error");
23
24      exit(0);
25  }
```

code/ecf/waitpid1.c

图 8.15 使用 waitpid 函数回收僵死子进程

code/ecf/waitpid2.c

```
1   #include "csapp.h"
2   #define N 2
3
4   int main()
5   {
6       int status, i;
7       pid_t pid[N+1], retpid;
8
9       for (i = 0; i < N; i++)
10          if ((pid[i] = Fork()) == 0) /* child */
11              exit(100+i);
12
13      /* parent reaps N children in order */
14      i = 0;
15      while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
16          if (WIFEXITED(status))
17              printf("child %d terminated normally with exit status=%d\n",
18                     retpid, WEXITSTATUS(status));
19          else
20              printf("child %d terminated abnormally\n", retpid);
21      }
22
```

```
23        /* The only normal termination is if there are no more children */
24        if (errno != ECHILD)
25            unix_error("waitpid error");
26
27        exit(0);
28    }
```

code/ecf/waitpid2.c

图 8.16 使用 waitpid 按照僵死子进程创建的顺序回收它们

注意，程序不会按照某种特殊的顺序回收子进程。图 8.16 展示了我们可以如何用 waitpid 按照父进程创建子进程的相同顺序来回收图 8.15 中的子进程。

练习题 8.4

考虑下面的程序：

code/ecf/waitprob1.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int status;
6       pid_t pid;
7
8       printf("Hello\n");
9       pid = Fork();
10      printf("%d\n", !pid);
11      if (pid != 0) {
12          if (waitpid(-1, &status, 0) > 0) {
13              if (WIFEXITED(status) != 0)
14                  printf("%d\n", WEXITSTATUS(status));
15          }
16      }
17      printf("Bye\n");
18      exit(2);
19  }
```

code/ecf/waitprob1.c

A. 这个程序会产生多少输出行？

B. 这些输出行的一种可能的顺序是什么？

8.4.4 让进程休眠

sleep 函数将一个进程挂起一段时间。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

返回：还要休眠的秒数。

sleep 返回 0（如果请求的时间量已经到了），或者返回剩下的要休眠的秒数。后一种情况是可能的，例如当 sleep 函数被一个信号中断过早返回时。我们将在 8.5 节中详细讨论信号。

我们发现很有用的另一个函数是 pause 函数，该函数让调用函数休眠，直到该进程收到一个信号为止。

```
#include <unistd.h>

int pause(void);
```

总是返回-1。

练习题 8.5

编写一个 sleep 的包装函数，叫做 snooze，带有下面的接口：

```
unsigned int snooze(unsigned int secs);
```

snooze 函数和 sleep 函数的行为完全一样，除了会打印出一条信息来描述进程实际休眠了多长时间以外。

```
Slept for 4 of 5 secs.
```

8.4.5 加载并运行程序

execve 函数在当前进程的上下文中加载并运行一个新程序。

```
#include <unistd.h>

int execve(char *filename, char *argv[], char *envp);
```

若成功则不返回，若错误则返回-1。

execve 函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量列表 envp。只有当出现错误时，例如不能发现 filename，execve 才会返回到调用程序。所以，不像 fork 会一次调用返回两次，execve 调用一次并从不返回。

如图 8.17 所示，参数列表是用数据结构表示的。argv 变量指向一个以 null 结尾的指针数组，其中每个指针都指向一个参数串。按照习俗，argv[0]是可执行目标文件的名字。环境变量的列表是由一个类似的数据结构表示的，如图 8.18 所示。envp 变量指向一个以 null 结尾的指针数组，其中每个指针指向一个环境变量串，其中每个串都是形如“NAME=VALUE”的名字—值对。

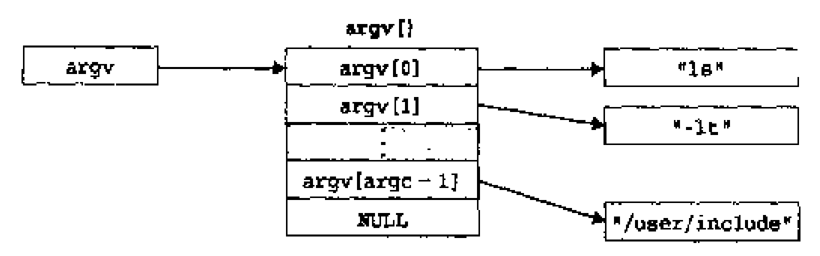


图 8.17 参数列表的组织结构

在 execve 加载了 filename 之后，它调用 7.9 节中描述的启动代码。启动代码准备好栈，并将控制传递给新程序的主函数，该主函数有如下形式的原型：

```
int main(int argc, char **argv, char **envp);
```

或者等价的：

```
int main(int argc, char *argv[], char *envp[]);
```

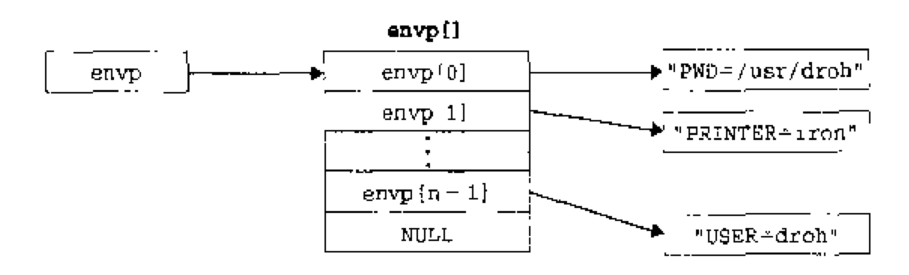


图 8.18　环境变量列表的组织结构

当 main 开始在一个 Linux 系统上执行时，用户栈有如图 8.19 所示的组织。让我们从栈底（高地址）往栈顶（低地址），依次看一看。首先是参数和环境字符串，它们都是连续地存放在栈中的，一个接一个，没有分隔。紧随其后，在栈的更上层里，是以 null 结尾的指针数组，其中每个指针都指向栈中的一个环境变量串。全局变量 environ 指向这些指针中的第一个 envp[0]。紧随环境变量数组其后的是以 null 结尾的 argv[ ]数组，其中每个元素都指向栈中一个参数串。在栈的顶部是 main 函数的 3 个参数：envp，它指向 envp[ ]数组；argv，它指向 argv[ ]数组；argc，它给出 argv[ ]中非空指针的数量。

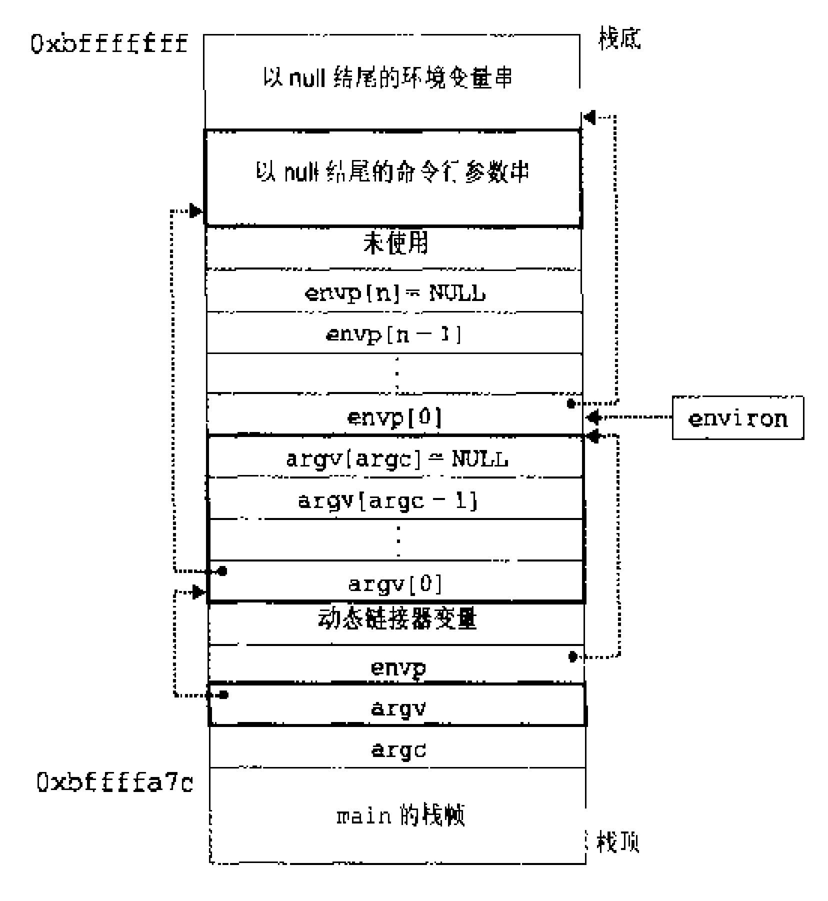


图 8.19　当一个新的程序开始时，用户栈的典型组织

Unix 提供了几个函数来操作环境数组：

```
#include <stdlib.h>

char *getenv(const char *name);
```
返回：若存在则为指向名字的指针，若无匹配的，则为 NULL.

getenv 函数在环境数组中搜索字符串“name=value”。如果找到了，它就返回一个指向 value 的指针，否则它就返回空指针。

```
#include <stdlib.h>

int setenv(const char *name, const char *newvalue, int overwrite);
```
返回：若成功则为 0，若错误则为 -1。

```
void unsetenv(const char *name);
```
返回：无。

如果环境数组包含一个形如“name=oldvalue”的字符串，那么 unsetenv 会删除它，而 setenv 会用 newvalue 代替 oldvalue，但是只有在 overwirte 非零时才会这样。如果 name 不存在，那么 setenv 就把“name=newvalue”添加到数组中。

旁注：在 Solaris 系统中设置环境变量

Solaris 提供 putenv 函数，而不是 setenv 函数。它并不提供相当于 unsetenv 函数功能的函数。

旁注：程序与进程

这是一个适当的地方，停下来，确认一下你是否理解了程序和进程之间的区别。程序是代码和数据的集合；程序可以作为目标模块存在于磁盘上，或者作为段存在于地址空间中。进程是执行中程序的一个特殊实例；程序总是运行在某个进程的上下文中。如果你想要理解 fork 和 execve 函数，理解这个差异是很重要的。fork 函数在新的子进程中运行相同的程序，新的子进程是父进程的一个复制品。execve 函数在当前进程的上下文中加载并运行一个新的程序。它会覆盖当前进程的地址空间，但并没有创建一个新进程。新的程序仍然有相同的 PID，并且继承了调用 execve 函数时打开的所有文件描述符。

练习题 8.6

编写一个叫做 myecho 的程序，它打印出它的命令行参数和环境变量。例如：

```
unix> ./myecho arg1 arg2
Command line arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2
Environment variables:
    envp[ 0]: PWD=/usr0/droh/ics/code/ecf
    envp[ 1]: TERM=emacs
...
```

```
envp[25]: USER=droh
envp[26]: SHELL=/usr/local/bin/tcsh
envp[27]: HOME=/usr0/droh
```

8.4.6 利用 fork 和 execve 运行程序

像 Unix shell 和 Web 服务器（第 12 章）这样的程序大量使用了 fork 和 execve 函数。shell 是一个交互型的应用级程序，它代表用户运行其他程序。最早的 shell 是 sh 程序，后面出现了一些变种，比如 csh、tcsh、ksh 和 bash。shell 执行一系列的读/求值（read/evaluate）步骤，然后终止。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。

图 8.20 展示了一个简单 shell 的 main 例程。shell 打印一个命令行提示符，等待用户在 stdin 上输入命令行，然后求值（evaluate）这个命令行。

code/ecf/shellex.c

```
1   #include "csapp.h"
2   #define MAXARGS 128
3
4   /* function prototypes */
5   void eval(char *cmdline);
6   int parseline(const char *cmdline, char **argv);
7   int builtin_command(char **argv);
8
9   int main()
10  {
11      char cmdline[MAXLINE]; /* command line */
12
13      while (1) {
14          /* read */
15          printf("> ");
16          Fgets(cmdline, MAXLINE, stdin);
17          if (feof(stdin))
18              exit(0);
19
20          /* evaluate */
21          eval(cmdline);
22      }
23  }
```

code/ecf/shellex.c

图 8.20 一个简单 shell 程序的 main 例程

图 8.21 展示了求值（evaluate）命令行的代码。它的第一个任务是调用 parseline 函数（图 8.22），这个函数解析了以空格分隔的命令行参数，并构造最终会传递给 execve 的 argv 向量。第一个参数被假设为要么是一个内置的 shell 命令名字，马上就会解释这个命令，要么是一个可执行的目标文件，会在一个新的子进程的上下文中加载并运行这个文件。

code/ecf/shellex.c

```
1   /* eval - evaluate a command line */
2   void eval(char *cmdline)
```

```
3   {
4       char *argv[MAXARGS];    /* argv for execve() */
5       int bg;                 /* should the job run in bg or fg? */
6       pid_t pid;              /* process id */
7
8       bg = parseline(cmdline, argv);
9       if (argv[0] == NULL)
10          return;  /* ignore empty lines */
11
12       if (!builtin_command(argv)) {
13          if ((pid = Fork()) == 0) {  /* child runs user job */
14              if (execve(argv[0], argv, environ) < 0) {
15                  printf("%s: Command not found.\n", argv[0]);
16                  exit(0);
17              }
18          }
19
20          /* parent waits for foreground job to terminate */
21          if (!bg) {
22          int status;
23          if (waitpid(pid, &status, 0) < 0)
24              unix_error("waitfg: waitpid error");
25          }
26          else
27              printf("%d %s", pid, cmdline);
28      }
29      return;
30  }
31
32  /* if first arg is a builtin command, run it and return true */
33  int builtin_command(char **argv)
34  {
35      if (!strcmp(argv[0], "quit"))       /* quit command */
36          exit(0);
37      if (!strcmp(argv[0], "&"))          /* ignore singleton & */
38          return 1;
39      return 0;                           /* not a builtin command */
40  }
```

code/ecf/shellex.c

图 8.21　eval：求值（evaluate）shell 命令行

code/ecf/shellex.c

```
1   /* parseline - parse the command line and build the argv array */
2   int parseline(const char *cmdline, char **argv)
3   {
4       char array[MAXLINE]; /* holds local copy of command line */
5       char *buf = array;   /* ptr that traverses command line */
6       char *delim;         /* points to first space delimiter */
7       int argc;            /* number of args */
```

```
8       int bg;                    /* background job? */
9
10      strcpy(buf, cmdline);
11      buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
12      while (*buf && (*buf == ' ')) /* ignore leading spaces */
13          buf++;
14
15      /* build the argv list */
16      argc = 0;
17      while ((delim = strchr(buf, ' '))) {
18          argv[argc++] = buf;
19          *delim = '\0';
20          buf = delim + 1;
21          while (*buf && (*buf == ' ')) /* ignore spaces */
22                 buf++;
23      }
24      argv[argc] = NULL;
25
26      if (argc == 0)  /* ignore blank line */
27          return 1;
28
29      /* should the job run in the background? */
30      if ((bg = (*argv[argc-1] == '&')) != 0)
31          argv[--argc] = NULL;
32
33      return bg;
34  }
```

code/ecf/shellex.c

图8.22 parseline：为 shell 一个输入行

如果最后一个参数是一个“&”字符，那么 parseline 返回 1，表示应该在后台执行该程序（shell 不会等待它完成）。否则，它返回 0，表示应该在前台执行这个程序（shell 会等待它完成）。

在解析了命令行之后，eval 函数调用 builtin_command 函数，该函数检查第一个命令行参数是否是一个内置的 shell 命令。如果是，它就立即解释这个命令，并返回值 1。否则，返回 0。我们简单的 shell 只有一个内置命令——quit 命令，该命令是用来终止 shell 的。实际使用的 shell 有大量的命令，比如 pwd、jobs 和 fg。

如果 builtin_command 返回 0，那么 shell 创建一个子进程，并在子进程中执行所请求的程序。如果用户要求在后台运行该程序，那么 shell 返回到循环的顶部，等待下一个命令行。否则，shell 使用 waitpid 函数等待作业的终止。当作业终止时，shell 就开始下一轮迭代。

注意，这个简单的 shell 是有缺陷的，因为它并不回收它的后台子进程。修改这个缺陷就要求使用信号，我们将在下一节中讲述信号。

8.5 信号

到目前为止，在我们对异常控制流的学习中，我们已经看到了硬件和软件是如何合作以提供基

本的低层异常机制的。我们也看到了操作系统是如何利用异常来支持更高层形式的异常控制流的，也就是所谓的上下文切换。在本节中，我们将研究一个更高层软件形式的异常，称为 Unix 信号，它允许进程中断其他进程。

一个信号(signal)就是一条消息，它通知进程一个某种类型的事件已经在系统中发生了。比如，图 8.23 展示了 Linux 系统上支持的 30 种不同类型的信号。

号码	名字	默认行为	相应事件
1	SIGHUP	终止	终端线挂起
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储存储器（1）	跟踪陷阱
6	SIGABRT	终止并转储存储器（1）	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储存储器（1）	浮点异常
9	SIGKILL	终止（2）	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储存储器（1）	无效的存储器引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程暂停或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT（2）	不来自终端的暂停信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的暂停信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

图 8.23 Linux 信号

其他 Unix 系统是类似的。注意：（1）多年前，主存储器是用一种称为磁芯存储器（core memory）的技术来实现的。“转储存储器（dumping core）”是一个历史术语，意思是把代码和数据存储器段的映像写到磁盘上。（2）这个信号既不能被捕获，也不能被忽略。

每种信号类型都对应于某个类型的系统事件。低层的硬件异常是由内核异常处理程序处理的，对用户进程而言通常是不可见的。信号提供了一种机制向用户进程通知这些异常的发生。比如，如果一个进程试图除以0，那么内核就发送给它一个SIGFPE信号（号码8）。如果一个进程执行一条非法指令，那么内核就发送给它一个SIGILL信号（号码4）。如果进程有非法存储器引用，内核就发送给它一个SIGSEGV信号（号码11）。其他信号对应于内核或者其他用户进程中较高层的软件事件。比如，如果当进程在后台运行时，你键入ctrl-c（也就是同时按下ctrl键和c键），那么内核就会发送一个SIGINT信号（号码2）给前台进程。一个进程可以通过发送一个SIGKILL信号（号码9）强制终止另外一个进程。当一个子进程终止或者暂停时，内核会发送一个SIGCHLD信号（号码17）给父进程。

8.5.1 信号术语

传送一个信号到目的进程是由两个不同步骤组成的：

- *发送信号*。内核通过更新目的进程上下文中的某个状态，发送（递送）一个信号给目的进程。发送信号可以有如下两种原因：①内核检测到一个系统事件，比如除零错误或者子进程终止；②一个进程调用了kill函数（在下一节中讨论），显式地要求内核发送一个信号给目的进程。一个进程可以发送信号给它自己。
- *接收信号*。当目的进程被内核强迫以某种方式对信号的发送做出反应时，目的进程就接收了信号。进程可以忽略这个信号，终止，或者通过执行一个称为*信号处理程序*（signal handler）的用户层函数捕获这个信号。

一个只发出而没有被接收的信号叫做*待处理信号*（pending signal）。在任何时刻，一种类型至多只会有一个待处理信号。如果一个进程有一个类型为k的待处理信号，那么任何接下来发送到这个进程的类型为k的信号都不会排队等待，它们只是被简单地丢弃。一个进程可以有选择性地阻塞接收某种信号。当一种信号被阻塞时，它仍可以被发送，但是产生的待处理信号不会被接收，直到进程取消对这种信号的阻塞。

一个待处理信号最多只能被接收一次。内核为每个进程在pending位向量中维护着待处理信号集合，而在blocked位向量中维护着被阻塞的信号集合。只要一个类型为k的信号被传送，内核就会在pending位向量中设置第k个位，而只要一个类型为k的信号被接收，内核就会在pengding位向量中清除第k个位。

8.5.2 发送信号

Unix系统提供了大量的机制，用来发送信号给进程。所有这些机制都是基于进程组（process group）这个概念的。

进程组

每个进程都只属于一个进程组，进程组是由一个正整数进程组ID来标识的。getpgrp函数返回当前进程的进程组ID：

```
#include <unistd.h>

pid_t getpgrp(void);
```

返回：调用进程的进程组ID。

默认地，一个子进程和它的父进程同属于一个进程组。一个进程可以通过使用 setpgid 函数来改变自己或者其他进程的进程组：

```
#include <unistd.h>

pid_t setpgid(pid_t pid, pid_t pgid);
```

返回：若成功则为 0，若错误则为-1。

setpgid 函数将进程 pid 的进程组改为 pgid。如果 pid 是 0，那么就使用当前进程的 PID。如果 pgid 是 0，那么就用 pid 指定的进程的 PID 作为进程组 ID。例如，如果进程 15213 是调用进程，那么

```
setpgid(0, 0);
```

会创建一个新的进程组，其进程组 ID 是 15213，并且把进程 15213 加入到这个新的进程组中。

用 kill 程序发送信号

/bin/kill 程序可以向另外的进程发送任意的信号。比如，命令

```
unix> kill .9 15213
```

发送信号 9（SIGKILL）给进程 15213。一个为负的 PID 会导致信号被发送到 PID 进程组中的每个进程。比如，命令

```
unix> kill .9 .15213
```

发送一个 SIGKILL 信号给进程组 15213 中的每个进程。

从键盘发送信号

Unix shell 使用作业（job）的抽象概念来表示求值（evaluating）一条命令行而产生的进程。在任何时刻，至多只有一个前台作业和 0 个或多个后台作业。比如，键入

```
unix> ls | sort
```

创建一个由两个进程组成的前台作业，这两个进程是通过 Unix 管道连接起来的：一个进程运行 ls 程序，另一个运行 sort 程序。

shell 为每个作业创建一个独立的进程组。典型地，进程组 ID 是取自作业中父进程中的一个。比如，图 8.24 展示了一个有一个前台作业和两个后台作业的 shell。前台作业中的父进程 PID 为 20，进程组 ID 也为 20。父进程创建两个子进程，每个也都是进程组 20 的成员。

在键盘上输入 ctrl-c，发送 SIGINT 信号到 shell。shell 捕获该信号（参见 8.5.3 节），然后发送 SIGINT 信号到这个前台进程组中的每个进程。在默认情况中，结果是终止前台作业。类似地，输入 ctrl-z 会发送一个 SIGTSTP 信号到 shell，shell 捕获这个信号，并发送 SIGTSTP 信号给前台进程组中的每个进程。在默认情况下，结果是暂停（挂起）前台作业。

用 kill 函数发送信号

进程通过调用 kill 函数发送信号给其他进程（包括它们自己）：

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

返回：若成功则为 0，若错误则为-1。

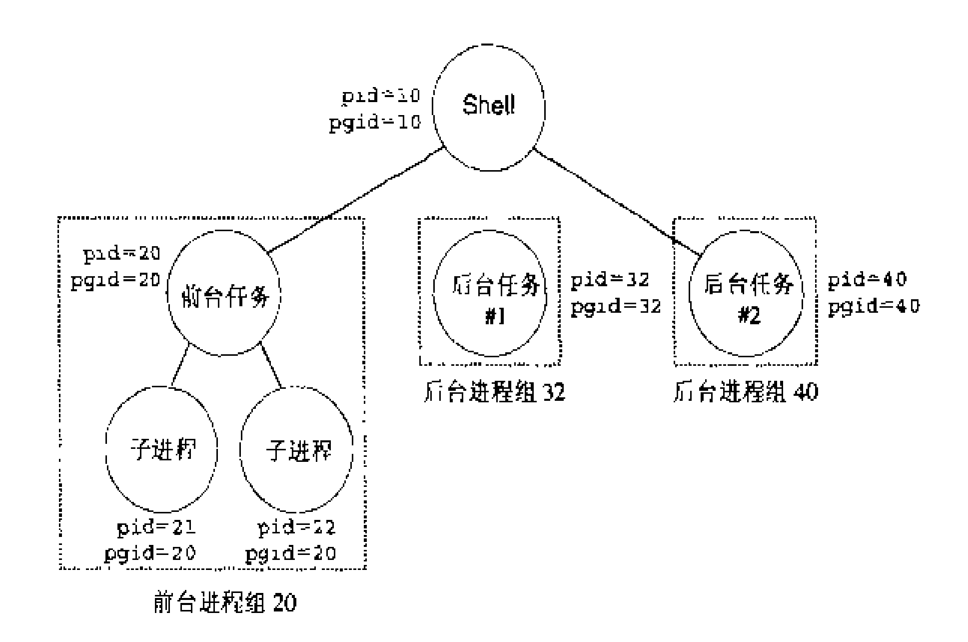


图 8.24 前台和后台进程组

如果 pid 大于零，那么 kill 函数发送信号号码 sig 给进程 pid。如果 pid 小于零，那么 kill 发送信号 sig 给进程组 abs(pid)中的每个进程。图 8.25 展示了一个示例，父进程利用 kill 函数发送 SIGKILL 信号给它的子进程。

code/ecf/kill.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6
7       /* child sleeps until SIGKILL signal received, then dies */
8       if ((pid = Fork()) == 0) {
9           Pause();  /* wait for a signal to arrive */
10          printf("control should never reach here!\n");
11          exit(0);
12      }
13
14      /* parent sends a SIGKILL signal to a child */
15      Kill(pid, SIGKILL);
16      exit(0);
17  }
```

code/ecf/kill.c

图 8.25 使用 kill 函数传递信号给子进程

用 alarm 函数发送信号

进程可以通过调用 alarm 函数向它自己发送 SIGALRM 信号：

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

返回：前一次闹钟剩余的秒数，若以前没有设定闹钟，则为 0。

alarm 函数安排内核在 secs 秒内发送一个 SIGALRM 信号给调用进程。如果 secs 是零，那么不会调度新的闹钟（alarm）。在任何情况中，对 alarm 的调用都将取消任何待处理的（pending）闹钟，并且返回任何待处理的闹钟应该发送前剩下的秒数（如果这次对 alarm 的调用没有取消它的话），或者如果没有任何待处理的闹钟，就返回零。

图 8.26 展示了一个调用 alarm 的程序，它安排自己被 SIGALRM 信号在 5 秒内每秒中断一次。当传送第 6 个 SIGALRM 信号时，它就终止。

code/ecf/alarm.c

```
1   #include "csapp.h"
2
3   void handler(int sig)
4   {
5       static int beeps = 0;
6
7       printf("BEEP\n");
8       if (++beeps < 5)
9           Alarm(1); /* next SIGALRM will be delivered in 1s */
10      else {
11          printf("BOOM!\n");
12          exit(0);
13      }
14  }
15
16  int main()
17  {
18      Signal(SIGALRM, handler); /* install SIGALRM handler */
19      Alarm(1); /* next SIGALRM will be delivered in 1s */
20
21      while (1) {
22          ; /* signal handler returns control here each time */
23      }
24      exit(0);
25  }
```

code/ecf/alarm.c

图 8.26 使用 alarm 函数调度周期性事件

当我们运行图 8.26 中的程序时，我们得到以下的输出：5 秒内每秒一个“BEEP”，后面跟随着程序终止时的一个“BOOM”：

```
unix> ./alarm
```

```
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

注意图 8.26 中的程序使用 signal 函数设置了一个信号处理函数（handler），只要进程收到一个 SIGALRM 信号，就异步地调用该函数，中断 main 程序中的无限 while 循环。当 handler 返回时，控制传递回 main 函数，它就从当初被信号到达时中断了的地方继续执行。设置和使用信号处理程序可能是相当微妙的，这将是下面三节讨论的主题。

8.5.3 接收信号

当内核从一个异常处理程序返回，准备将控制传递给进程 p 时，它会检查未被阻塞的待处理信号的集合（pending&~blocked）。如果这个集合为空（通常情况），那么内核传递控制给 p 的逻辑控制流中的下一条指令（I_{next}）。

然而，如果集合是非空的，那么内核选择集合中的某个信号 k（通常是最小的 k），并且强制 p 接收信号 k。收到这个信号会触发进程的某种行为。一旦进程完成了这个行为，那么控制就传递回 p 的逻辑控制流中的下一条指令（I_{next}）。每个信号类型都有一个预定义的默认行为：

- 进程终止。
- 进程终止并转储存储器（dump core）。
- 进程暂停直到被 SIGCONT 信号重启。
- 进程忽略该信号。

图 8.23 展示了与每个信号类型相关联的默认行为。比如，收到 SIGKILL 的默认行为就是终止接收进程。另外，接收 SIGCHLD 的默认行为就是忽略这个信号。进程可以通过使用 signal 函数修改和信号相关联的默认行为。惟一的例外是 SIGSTOP 和 SIGKILL，它们的默认行为是不能修改的。

```
#include <signal.h>
typedef void handler_t(int)

handler_t *signal(int signum, handler_t *handler)
```

返回：若成功则为指向前次处理程序的指针，若出错则为 SIG-ERR 不设置 errno。

signal 函数可以通过下列三种方法之一来改变和信号 signum 相关联的行为：

- 如果 handler 是 SIG_IGN，那么忽略类型为 signum 的信号。
- 如果 handler 是 SIG_DFL，那么类型为 signum 的信号行为恢复为默认行为。
- 否则，handler 就是用户定义的函数的地址，称为*信号处理程序*（signal handler）。只要进程接收到一个类型为 signum 的信号，就会调用这个程序。通过把处理程序的地址传递到 signal 函数从而改变默认行为，这叫做*设置信号处理程序*。信号处理程序的调用被称为*捕捉信号*。信号处理程序的执行被称为*处理信号*。

当一个进程捕捉了一个类型为 k 的信号时，为信号 k 设置的处理程序被调用，同时惟一一个整数参数被设置为 k。这个参数允许同一个处理函数捕捉不同类型的信号。

当处理程序执行它的 return 语句时，控制（通常）传递回控制流中进程被信号接收中断位置处的指令。我们说“通常”是因为在某些系统中，被中断的系统调用会立即返回一个错误。

图 8.27 展示了一个捕获用户在键盘上输入 ctrl-c 时 shell 发送的 SIGINT 信号的程序。SIGINT 的默认行为是立即终止该进程。在这个示例中，我们将默认行为修改为捕捉信号，输出一条信息，然后终止该进程。

code/ecf/sigint1.c

```
1   #include "csapp.h"
2
3   void handler(int sig) /* SIGINT handler */
4   {
5       printf("Caught SIGINT\n");
6       exit(0);
7   }
8
9   int main()
10  {
11      /* Install the SIGINT handler */
12      if (signal(SIGINT, handler) == SIG_ERR)
13          unix_error("signal error");
14
15      pause(); /* wait for the receipt of a signal */
16
17      exit(0);
18  }
```

code/ecf/sigint1.c

图 8.27 一个捕捉 SIGINT 信号的程序

处理程序函数定义在第 3～7 行中。主函数在第 12～13 行设置处理程序，然后进入休眠状态，直到接收到一个信号（第 15 行）。当收到 SIGINT 信号时，运行处理程序，输出一条信息（第 5 行），然后终止这个进程（第 6 行）。

练习题 8.7

编写一个叫做 snooze 的程序，有一个命令行参数，用这个参数调用习题 8.5 中的 snooze 函数，然后终止。编写程序，使得用户可以通过在键盘上输入 ctrl-c 中断 snooze 函数。比如：

```
unix> ./snooze 5
Slept for 3 of 5 secs.   User hits crtl-c after 3 seconds
unix>
```

8.5.4 信号处理问题

对于只捕捉一个信号并终止的程序来说，信号处理是简单直接的。然而，当一个程序要捕捉多个信号时，一些细微的问题就产生了：

- 待处理信号被阻塞。Unix 信号处理程序典型地会阻塞当前处理程序正在处理的类型的待处

理信号。比如，假设一个进程捕捉了一个 SIGINT 信号，并且当前正在运行它的 SIGINT 处理程序。如果另一个 SIGINT 信号传递到这个进程，那么这个 SIGINT 将变成待处理的，但是不会被接收，直到处理程序返回。

- **待处理信号不会排队等待。** 任意类型至多只有一个待处理信号。因此，如果有两个类型为 k 的信号传送到一个目的进程，而由于目的进程当前正在执行信号 k 的处理程序，所以信号 k 是阻塞的，那么第二个信号就被简单地丢弃，它不会排队等待。关键思想是存在一个待处理的信号仅仅表明至少已经到达了一个信号。
- **系统调用可以被中断。** 像 read、write 和 accept 这样的系统调用潜在地会阻塞进程一段较长的时间，称之为*慢速系统调用*。在某些系统中，当处理程序捕捉到一个信号时，被中断的慢速系统调用在信号处理程序返回时不再继续，而是立即返回给用户一个错误条件，并将 errno 设置为 EINTR。

让我们利用一个简单的应用程序更深入地看看信号处理的细微之处，这个应用程序本质上类似于 shell 和 Web 服务器这样的真实程序。基本的结构是一个父进程创建一些子进程，这些子进程独立运行一会儿，然后终止。父进程必须回收子进程，以避免在系统中留下僵死进程。但是我们也想让父进程在子进程运行时可以自由地做其他工作。所以，我们决定用 SIGCHLD 处理程序回收子进程，而不是显式地等待子进程终止。（回想一下：只要子进程终止或者暂停时，内核就会发送一个 SIGCHLD 信号给父进程。）

图 8.28 展示了我们的第一次尝试。父进程设置了一个 SIGCHLD 处理程序，然后创建了三个子进程，其中每个子进程运行 1 秒，然后终止。同时，父进程等待来自终端的一个输入行，随后处理它。这个处理被模型化为一个无限循环。当每个子进程终止时，内核通过发送一个 SIGCHLD 信号通知父进程。父进程捕捉这个 SIGCHLD 信号，回收一个子进程，做一些其他的清除工作（模型化为 sleep(2)语句），然后返回。

code/ecf/signal1.c

```
1   #include "csapp.h"
2
3   void handler1(int sig)
4   {
5       pid_t pid;
6
7       if ((pid = waitpid(-1, NULL, 0)) < 0)
8           unix_error("waitpid error");
9       printf("Handler reaped child %d\n", (int)pid);
10      Sleep(2);
11      return;
12  }
13
14  int main()
15  {
16      int i, n;
17      char buf[MAXBUF];
```

```
18
19      if (signal(SIGCHLD, handler1) == SIG_ERR)
20          unix_error("signal error");
21
22      /* parent creates children */
23      for (i = 0; i < 3; i++) {
24          if (Fork() == 0) {
25              printf("Hello from child %d\n", (int)getpid());
26              Sleep(1);
27              exit(0);
28          }
29      }
30
31      /* parent waits for terminal input and then processes it */
32      if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
33          unix_error("read");
34
35      printf("Parent processing input\n");
36      while (1)
37          ;
38
39      exit(0);
40  }
```

code/ecf/signal1.c

图 8.28 signal1

这个程序是有缺陷的，因为它无法处理信号会阻塞、信号不会排队等待和系统调用可以被中断这些情况。

图 8.28 中的 signal1 程序看起来相当简单。然而，当我们在 Linux 系统上运行它时，我们得到如下输出：

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<cr>
Parent processing input
```

从输出中，我们注意到，尽管发送了 3 个 SIGCHLD 信号给父进程，但是只有其中的两个信号被接收了，因此父进程只是回收了两个子进程。如果我们挂起父进程，我们看到，实际上，子进程 10321 没有被回收，成为了一个僵死进程：

```
<ctrl-z>
Suspended
```

```
linux> ps
PID TTY STAT TIME COMMAND
...
10319 p5 T      0:03 signal1
10321 p5 Z      0:00 (signal1 <zombie>)
10323 p5 R      0:00 ps
```

哪里出错了呢？问题就在于我们的代码没有解决信号可以阻塞和不会排队等待这样的事实。下面是所发生的情况：

父进程接收并捕捉了第一个信号。当处理程序还在处理第一个信号时，第二个信号就传送并添加到了待处理信号集合里。然而，因为 SIGCHLD 信号被 SIGCHLD 处理程序阻塞了，所以第二个信号就不会被接收。此后不久，就在处理程序还在处理第一个信号时，第三个信号到达了。因为已经有了一个待处理的 SIGCHLD，第三个 SIGCHLD 信号会被丢弃。一段时间之后，处理程序返回，内核注意到有一个待处理的 SIGCHLD 信号，就迫使父进程接收这个信号。父进程捕获信号，并第二次执行处理程序。在处理程序完成对第二个信号的处理之后，已经没有待处理的 SIGCHLD 信号了，而且也绝不会再有，因为第三个 SIGCHLD 的所有信息都已经丢失了。由此得到的重要教训是，信号不可以用来对其他进程中发生的事件计数。

为了修正这个问题，我们必须回想一下，存在一个待处理的信号只是暗示自进程最后一次收到一个信号以来，至少已经有一个这种类型的信号被发送了。所以我们必须修改 SIGCHLD 处理程序，使每次 SIGCHLD 处理程序被调用时，回收尽可能多的僵死子进程。图 8.29 展示了修改后的 SIGCHLD 处理程序。

code/ecf/signal2.c

```
1   #include "csapp.h"
2
3   void handler2(int sig)
4   {
5       pid_t pid;
6
7       while ((pid = waitpid(-1, NULL, 0)) > 0)
8           printf("Handler reaped child %d\n", (int)pid);
9       if (errno != ECHILD)
10          unix_error("waitpid error");
11      Sleep(2);
12      return;
13  }
14
15  int main()
16  {
17      int i, n;
18      char buf[MAXBUF];
19
20      if (signal(SIGCHLD, handler2) == SIG_ERR)
```

```
21              unix_error("signal error");
22
23          /* parent creates children */
24          for (i = 0; i < 3; i++) {
25              if (Fork() == 0) {
26                  printf("Hello from child %d\n", (int)getpid());
27                  Sleep(1);
28                  exit(0);
29              }
30          }
31
32          /* parent waits for terminal input and then processes it */
33          if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
34              unix_error("read error");
35
36          printf("Parent processing input\n");
37          while (1)
38              ;
39
40          exit(0);
41      }
```

code/ecf/signal2.c

图 8.29 signal2

图 8.28 的一个改进版本，它能够正确解决信号会阻塞和不会排队等待的情况。然而，它没有考虑系统调用被中断的可能性。

当我们在 Linux 系统上运行 signal2 时，它可以正确地回收所有的僵死子进程了。

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
<cr>
Parent processing input
```

然而，我们还没有完成任务。如果我们在 Solaris 系统上运行 signal2 程序，它会正确地回收所有的僵死子进程。然而，现在，被阻塞的 read 系统调用在我们在键盘上进行输入之前，提前返回一个错误：

```
solaris> ./signal2
Hello from child 18906
Hello from child 18907
Hello from child 18908
```

```
Handler reaped child 18906
Handler reaped child 18908
Handler reaped child 18907
read: Interrupted system call
```

出了什么问题呢？出现这个问题是因为在这个 Solaris 系统上，诸如 read 这样的慢速系统调用在被信号发送中断后，是不会自动重启的。相反地，和 Linux 系统自动重启被中断的系统调用不同，它们会提前返回给调用应用程序一个错误条件。

为了编写可移植的信号处理代码，我们必须考虑系统调用过早返回，然后手动重启它们的情况。图 8.30 展示了对 signal1 的修改，它会手动地重启被终止的 read 调用。errno 中的 EINTR 返回代码表明 read 系统调用在它被中断后提前返回了。

code/ecf/signal3.c

```
1    #include "csapp.h"
2
3    void handler2(int sig)
4    {
5        pid_t pid;
6
7        while ((pid = waitpid(-1, NULL, 0)) > 0)
8            printf("Handler reaped child %d\n", (int)pid);
9        if (errno != ECHILD)
10           unix_error("waitpid error");
11       Sleep(2);
12       return;
13   }
14
15   int main() {
16       int i, n;
17       char buf[MAXBUF];
18       pid_t pid;
19
20       if (signal(SIGCHLD, handler2) == SIG_ERR)
21           unix_error("signal error");
22
23       /* parent creates children */
24       for (i = 0; i < 3; i++) {
25           pid = Fork();
26           if (pid == 0) {
27               printf("Hello from child %d\n", (int)getpid());
28               Sleep(1);
29               exit(0);
30           }
31       }
32
```

```
33          /* Manually restart the read call if it is interrupted */
34          while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35              if (errno != EINTR)
36                  unix_error("read error");
37
38          printf("Parent processing input\n");
39          while (1)
40              ;
41
42          exit(0);
43      }
```

code/ecf/signal3.c

图 8.30 signal3

图 8.29 的一个改进版本，它正确地解决了系统调用可能被中断的情况。

当我们在一台 Solaris 系统上运行我们新的 signal3 程序时，程序会正确运行：

```
solaris> ./signal3
Hello from child 19571
Hello from child 19572
Hello from child 19573
Handler reaped child 19571
Handler reaped child 19572
Handler reaped child 19573
<cr>
Parent processing input
```

8.5.5 可移植的信号处理

不同系统之间，信号处理语义的差异——比如一个中断慢速系统调用是否重启或者永久放弃——是 Unix 信号处理的一个缺陷。为了处理这个问题，Posix 标准定义了 sigaction 函数，它允许 Posix 兼容系统的用户，比如 Linux 和 Solaris 的用户，显式地指定他们想要的信号处理语义。

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
```

返回：若成功则为 0，若出错则为 -1。

sigaction 函数运用并不广泛，因为它要求用户设置一个结构的项目（entry）。一个更简洁的方式，最初是 Stevens 提出的[81]，就是定义一个包装（wrapper）函数，称为 Signal，它为我们调用 sigaction。图 8.31 给出了 Signal 的定义，它的调用方式与 signal 函数的调用方式一样。Signal 包装函数设置了一个信号处理程序，其信号处理语义如下：

- 只有这个处理程序当前正在处理的那种类型的信号被阻塞。
- 和所有信号实现一样，信号不会排队等待。

- 只要可能，被中断的系统调用会自动重启。
- 一旦设置了信号处理程序，它就会一直保持，直到Signal带着handler参数为SIG_IGN或者SIG_DFL被调用。（一些比较老的Unix系统会在一个处理程序处理完一个信号之后，将信号行为恢复为它的默认行为。）

code/src/csapp.c

```
1   handler_t *Signal(int signum, handler_t *handler)
2   {
3       struct sigaction action, old_action;
4
5       action.sa_handler = handler;
6       sigemptyset(&action.sa_mask); /* block sigs of type being handled */
7       action.sa_flags = SA_RESTART; /* restart syscalls if possible */
8
9       if (sigaction(signum, &action, &old_action) < 0)
10          unix_error("Signal error");
11      return (old_action.sa_handler);
12  }
```

code/src/csapp.c

图8.31 Signal

sigaction的一个包装函数，它提供Posix兼容系统的可移植信号处理。

图8.32展示了图8.29中signal2程序的一个版本，该版本使用我们的Signal包装函数在不同的计算机系统上获得可预测的信号处理语义。惟一的区别是我们是通过调用Signal而不是调用signal来设置处理程序的。现在，程序既可以在Solaris也可以在Linux系统上正确运行了，而我们也不再需要手动地重启被中断的read系统调用了。

code/ecf/signal4.c

```
1   #include "csapp.h"
2
3   void handler2(int sig)
4   {
5       pid_t pid;
6
7       while ((pid = waitpid(-1, NULL, 0)) > 0)
8           printf("Handler reaped child %d\n", (int)pid);
9       if (errno != ECHILD)
10          unix_error("waitpid error");
11      Sleep(2);
12      return;
13  }
14
15  int main()
16  {
17      int i, n;
18      char buf[MAXBUF];
19      pid_t pid;
```

```
20
21          Signal(SIGCHLD, handler2); /* sigaction error-handling wrapper */
22
23          /* parent creates children */
24          for (i = 0; i < 3; i++) {
25              pid = Fork();
26              if (pid == 0) {
27                  printf("Hello from child %d\n", (int)getpid());
28                  Sleep(1);
29                  exit(0);
30              }
31          }
32
33          /* parent waits for terminal input and then processes it */
34          if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
35              unix_error("read error");
36
37          printf("Parent processing input\n");
38          while (1)
39              ;
40          exit(0);
41  }
```

code/ecf/signal4.c

图 8.32 signal4

图 8.29 的一个版本，该版本通过使用我们的 Signal 包装函数得到可移植的信号处理语义。

8.5.6 显式地阻塞信号

应用程序可以使用 sigpromask 函数显式地阻塞和取消阻塞选择的信号：

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

返回：如果成功则为 0，若出错则为 -1。

```
int sigismember(const sigset_t *set, int signum);
```

返回：若 signum 是 sef 的成员则为 1，若出错则为 0。

sigpromask 函数改变当前已阻塞信号的集合（blocked 位向量在 8.5.1 节中描述）。具体的行为依赖于 how 的值：

- SIG_BLOCK：添加 set 中的信号到 blocked 中（blocked=blocked|set）。
- SIG_UNBLOCK：从 blocked 中删除 set 中的信号（blocked=blocked&~set）。
- SIG_SETMASK：blocked=set。

如果 oldset 非空，blocked 位向量以前的值会保存在 oldset 中。

可以使用下列函数操作像 set 这样的信号集合。sigemptyset 初始化 set 为空集。sigfillset 函数将每个信号添加到 set 中。sigaddset 函数添加 signum 到 set，sigdelset 从 set 中删除 signum，如果 signum 是 set 的成员，那么 sigismember 返回 1，反之则返回 0。

sigprocmask 函数对于同步父子进程是很方便的。比如，考虑图 8.33，它总结了一个典型的 Unix shell 的结构。父进程在一个作业列表中记录着它的子进程。当父进程创建一个新的子进程时，它就把这个子进程添加到作业列表中。当父进程在 SIGCHLD 处理程序中回收一个终止的（僵死）子进程时，它就从作业列表中删除这个子进程。

code/ecf/procmask.c

```
1   void handler(int sig)
2   {
3     pid_t pid;
4     while ((pid = waitpid(-1, NULL, 0)) > 0) /* Reap a zombie child */
5         deletejob(pid); /* Delete the child from the job list */
6     if (errno != ECHILD)
7         unix_error("waitpid error");
8   }
9
10  int main(int argc, char **argv)
11  {
12      int pid;
13      sigset_t mask;
14
15      Signal(SIGCHLD, handler);
16      initjobs(); /* Initialize the job list */
17
18      while (1) {
19          Sigemptyset(&mask);
20          Sigaddset(&mask, SIGCHLD);
21          Sigprocmask(SIG_BLOCK,&mask, NULL); /* Block SIGCHLD */
22
23          /* Child process */
24          if ((pid = Fork()) == 0) {
25              Sigprocmask(SIG_UNBLOCK,&mask, NULL); /* Unblock SIGCHLD */
26              Execve("/bin/ls", argv, NULL);
27          }
28
29          /* Parent process */
30          addjob(pid); /* Add the child to the job list */
31          Sigprocmask(SIG_UNBLOCK,&mask, NULL); /* Unblock SIGCHLD */
32      }
33      exit(0);
34  }
```

code/ecf/procmask.c

图 8.33 用 sigprocmask 来同步进程

这个示例中，父进程在相应的 deletejob 之前保证执行了 addjob。

如果我们不以某种方法同步父子进程，那么就可能发生下面的情况：

- 父进程执行 fork 函数，并且内核调度新创建的子进程代替父进程运行。
- 在父进程可以再次运行之前，子进程会终止，并变成一个僵死进程，使得内核传递一个 SIGCHLD 信号给父进程。
- 后来，当父进程再次变成可运行但又在它执行之前，内核注意到待处理的 SIGCHLD 信号，并通过在父进程中运行处理程序接收这个信号。
- 处理程序回收终止的子进程，并调用 deletejob，这个函数什么也不做，因为父进程还没有把该子进程添加到列表中。
- 在处理程序运行完毕后，内核运行父进程，父进程从 fork 返回，通过调用 addjob 错误地把（不存在的）子进程添加到作业列表中。

关键问题是如果我们什么都不做，那么就可能在执行 addjob 之前，执行 deletejob。图 8.33 展示了改正这个问题的一种方法。通过在调用 fork 之前，阻塞 SIGCHLD 信号，然后在我们调用了 addjob 之后就取消阻塞这些信号，我们保证了在子进程被添加到作业列表中之后，回收该子进程。

注意子进程继承了它们父进程的被阻塞集合，所以我们必须在调用 execve 之前，小心地解除子进程中阻塞的 SIGCHLD 信号。

8.6 非本地跳转

C 提供了一种形式的用户级异常控制流，称为非本地跳转（nonlocal jump），它将控制直接从一个函数转移到另一个当前正在执行的函数，而不需要经过正常的调用-返回序列。非本地跳转是通过 setjmp 和 longjmp 函数来提供的。

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

返回：setjmp 返回 0，longjmp 返回非零。

setjmp 函数在 env 缓冲区中保存当前栈的内容，以供后面 longjmp 使用，并返回 0。

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

从不返回。

longjmp 函数从 env 缓冲区中恢复栈的内容，然后触发一个从最近一次初始化 env 的 setjmp 调用的返回。然后 setjmp 返回，并带有非零的返回值 retval。

第一眼看过去，setjmp 和 longjmp 之间的相互关系令人迷惑。setjmp 函数只被调用一次，但返回多次——一次是当第一次调用 setjmp，而栈的上下文保存在缓冲区 env 中时；一次是为每个相应的 longjmp。另一方面，longjmp 函数只被调用一次，但从不返回。

非本地跳转的一个重要应用就是允许从一个深层嵌套的函数调用中立即返回，通常是由检测到

某个错误情况引起的。如果在一个深层嵌套的函数调用中发现了一个错误情况，我们可以使用非本地跳转直接返回到一个普通的本地化的错误处理程序，而不是费力地解开调用栈。

图 8.34 展示了一个示例，说明这可能是如何工作的。main 函数首先调用 setjmp 以保存当前栈的上下文，然后调用函数 foo，foo 再调用函数 bar。如果 foo 或者 bar 遇到一个错误，它们立即通过一次 longjmp 调用从 setjmp 返回。setjmp 的非零返回值指明了错误类型，随后可以被解码，且在代码中的某个位置进行处理。

code/ecf/setjmp.c

```
1    #include "csapp.h"
2
3    jmp_buf buf;
4
5    int error1 = 0;
6    int error2 = 1;
7
8    void foo(void), bar(void);
9
10   int main()
11   {
12       int rc;
13
14       rc = setjmp(buf);
15       if (rc == 0)
16           foo();
17       else if (rc == 1)
18           printf("Detected an error1 condition in foo\n");
19       else if (rc == 2)
20           printf("Detected an error2 condition in foo\n");
21       else
22           printf("Unknown error condition in foo\n");
23       exit(0);
24   }
25
26   /* deeply nested function foo */
27   void foo(void)
28   {
29       if (error1)
30           longjmp(buf, 1);
31       bar();
32   }
33
34   void bar(void)
35   {
36       if (error2)
```

```
37          longjmp(buf, 2);
38  }
```

code/ecf/setjmp.c

图 8.34 非本地跳转的示例

这个示例表明了使用非本地跳转来从深层嵌套的函数调用中的错误情况恢复，而不需要解开整个栈的基本框架。

非本地跳转的另一个重要应用是使一个信号处理程序分支到一个特殊的代码位置，而不是返回到被信号到达中断了的指令的位置。图 8.35 展示了一个简单的程序，说明了这种基本技术。当用户在键盘上键入 ctrl-c 时，这个程序用信号和非本地跳转来实现软重启。sigsetjmp 和 siglongjmp 函数是 setjmp 和 longjmp 的可以被信号处理程序使用的版本。

code/ecf/restart.c

```
1   #include "csapp.h"
2
3   sigjmp_buf buf;
4
5   void handler(int sig)
6   {
7       siglongjmp(buf, 1);
8   }
9
10  int main()
11  {
12      Signal(SIGINT, handler);
13
14      if (!sigsetjmp(buf, 1))
15          printf("starting\n");
16      else
17          printf("restarting\n");
18
19      while(1) {
20          Sleep(1);
21          printf("processing... \n");
22      }
23      exit(0);
24  }
```

code/ecf/restart.c

图 8.35 当用户键入 ctrl-c 时，一个使用非本地跳转来重启动它本身的程序

在程序第一次启动时，对 sigsetjmp 函数的初始调用保存了栈和信号的上下文。随后，主函数进入一个无限处理循环。当用户键入 ctrl-c 时，shell 发送一个 SIGINT 信号给这个进程，该进程捕获这个信号。不是从信号处理程序返回，此时信号处理程序会将控制返回给被中断的处理循环，取而代之的是，处理程序执行一个非本地跳转，回到 main 函数的开始处。当我们在系统上运行这个程序时，我们得到以下输出：

```
unix> ./restart
starting
processing...
processing...
restarting          user hits ctrl-c
processing...
restarting          User hits ctrl-c
processing...
```

旁注：C++和 Java 中的软件异常

C++和 Java 提供的异常机制是较高层次的，是 C 的 setjmp 和 longjmp 函数的更加结构化的版本。你可以把 try 语句中的 catch 子句看做是 setjmp 函数的类似物。相似地，throw 语句就类似于 longjmp 函数。

8.7 操作进程的工具

Unix 系统提供了大量的监控和操作进程的有用工具：

strace: 打印一个程序和它的子进程调用的每个系统调用的轨迹。对于好奇的学生而言，这是一个令人着迷的工具。用-static 编译你的程序，能得到一个更清楚的轨迹，而不带有大量与共享库相关的输出。

ps: 列出系统中当前的进程（包括僵死进程）。

top: 打印出关于当前进程资源使用的信息。

kill: 发送一个信号给进程。对于调试带信号处理程序的程序以及清除难以琢磨的进程是非常有用的。

/proc（Linux 和 Solaris）：一个虚拟文件系统，以 ASCII 文本格式输出大量内核数据结构的内容，用户程序可以读取这些内容。比如，输入“cat /proc/loadavg”，观察在你的 Linux 系统上当前的平均负载。

8.8 小结

异常控制流发生在计算机系统的各个层次。在硬件层，异常是由处理器中的事件触发的控制流中的突变。控制流传递给一个软件处理程序，该处理程序进行一些处理，然后返回控制给被中断的控制流。

有四种不同类型的异常：中断、故障、终止和陷阱。当一个外部的 I/O 设备，例如定时器芯片或者一个磁盘控制器，设置了处理器芯片上的中断管脚时，（对于任意指令）中断会异步地发生。控制返回到中断指令[2]的下一条指令。执行一条指令可能导致故障和终止的发生。故障处理程序会重新开始故障指令，而终止处理程序从不将控制返回给被中断的流。最后，陷阱就像是用来实现系统调用的函数调用，系统调用提供给应用到操作系统代码的受控入口点。

2 原文为故障指令。——译者

在操作系统层，内核提供关于一个进程的基础性概念。一个进程提供给应用两个重要的抽象：①逻辑控制流，它提供给每个程序一个假象，好像它是在独占地使用处理器；②私有地址空间，它提供给每个程序一个假象，好像它是在独占地使用主存。

在操作系统和应用之间的接口处，应用可以创建子进程，等待它们的子进程暂停或者终止，运行新的程序，并捕捉来自其他进程的信号。信号处理的语义是微妙的，并且随系统不同而不同。然而，在与Posix兼容的系统上存在着一些机制，允许程序清楚地指定期望的信号处理语义。

最后，在应用层，C程序可以使用非本地跳转来规避正常的调用/返回栈规则，并且直接从一个函数分支到另一个函数。

参考文献说明

Intel 微体系结构规范包含对Intel处理器上的异常和中断的详细讨论[18]。操作系统教科书[70，75，83]包括关于异常、进程和信号的其他信息。Stevens的经典著作[76]，虽然有点过时了，但是仍然包含一些有价值的和可读性很高的描述，是关于如何在应用程序中处理进程和信号的。

家庭作业

8.8 ◆

在这一章里，我们介绍了一些有不寻常的调用和返回行为的函数：setjmp、longjmp、execve和fork。找到下列行为中和每个函数相匹配的一种：

A. 调用一次，返回两次。

B. 调用一次，从不返回。

C. 调用一次，返回一次或者多次。

8.9 ◆

下面程序的可能的输出是什么？

code/ecf/forkprob3.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int x = 3;
6
7       if (Fork() != 0)
8           printf("x=%d\n", ++x);
9
10      printf("x=%d\n", --x);
11      exit(0);
12  }
```

code/ecf/forkprob3.c

8.10 ◆

下面这个程序会输出多少个“hello”输出行？

code/ecf/forkprob5.c

```
1   #include "csapp.h"
2
3   void doit()
4   {
5       if (Fork() == 0) {
6           Fork();
7           printf("hello\n");
8           exit(0);
9       }
10      return;
11  }
12
13  int main()
14  {
15      doit();
16      printf("hello\n");
17      exit(0);
18  }
```

code/ecf/forkprob5.c

8.11 ◆

下面这个程序会输出多少个“hello”输出行？

code/ecf/forkprob6.c

```
1   #include "csapp.h"
2
3   void doit()
4   {
5       if (Fork() == 0) {
6           Fork();
7           printf("hello\n");
8           return;
9       }
10      return;
11  }
12
13  int main()
14  {
15      doit();
16      printf("hello\n");
17      exit(0);
18  }
```

code/ecf/forkprob6.c

8.12 ◆

下面这个程序的输出是什么？

code/ecf/forkprob7.c

```
1   #include "csapp.h"
2   int counter = 1;
3
4   int main()
5   {
6       if (fork() == 0) {
7           counter--;
8           exit(0);
9       }
10      else {
11          Wait(NULL);
12          printf("counter = %d\n", ++counter);
13      }
14      exit(0);
15  }
```

code/ecf/forkprob7.c

8.13 ◆

列举练习题 8.4 中程序所有可能的输出。

8.14 ◆◆

考虑下面的程序：

code/ecf/forkprob2.c

```
1   #include "csapp.h"
2
3   void end(void)
4   {
5       printf("2");
6   }
7
8   int main()
9   {
10      if (Fork() == 0)
11          atexit(end);
12      if (Fork() == 0)
13          printf("0");
14      else
15          printf("1");
16      exit(0);
17  }
```

code/ecf/forkprob2.c

判断下面哪个输出是可能的。注意：atexit 函数以一个指向函数的指针为输入，并将它添加到函数列表中（初始为空），当 exit 函数被调用时，会调用该列表中的函数。

A. 112002

B. 211020

C. 102120

D. 122001

E. 100212

8.15 ◆◆

使用 execve 编写一个叫做 myls 的程序，该程序的行为和/bin/ls 程序的一样。你的程序应该接受相同的命令行参数，解释同样的环境变量，并产生相同的输出。

ls 程序是从 COLUMNS 环境变量中获悉屏幕的宽度的。如果没有设置 COLUMNS，那么 ls 会假设屏幕宽 80 列。因此，你可以通过把 COLUMNS 环境设置得小于 80，来检查你对环境变量的处理。

```
unix> setenv COLUMNS 40
unix> ./myls
... output is 40 columns wide
unix> unsetenv COLUMNS
unix> ./myls
... output is now 80 columns wide
```

8.16 ◆◆◆

修改图 8.15 中的程序，以满足下面两个条件：

1. 每个子进程在试图写一个只读文本段中的位置时会异常终止。
2. 父进程打印和下面所示相同（除了 PID）的输出：

```
child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

提示：请参考 wait(2)和 psignal(3)的 man 页。

8.17 ◆◆◆

编写你自己版本的 Unix system 函数：

```
int mysystem(char *command);
```

mysystem 函数通过调用“/bin/sh -c command”来执行 command，然后在 command 完成后返回。如果 command 正常退出（通过调用 exit 函数或者执行一个 return 语句），那么 mysystem 返回 command 的退出状态。比如，如果 command 通过调用 exit(8)终止，那么 mysystem 返回值 8。否则，如果 command 异常终止，那么 mysystem 返回由 shell 返回的状态。

8.18 ◆

你的一位同事正在考虑使用信号来允许一个父进程计算子进程中发生的事件数。基本思路是每次一个事件发生时，通过发送一个信号来通知父进程，并且让父进程的信号处理程序增加全局变量

counter，父进程随后可以在子进程终止时检测该变量。然而，当在系统上运行图 8.36 中的测试程序时，他发现当父进程调用 printf 时，counter 总是保持一个值 2，即使是子进程已经发送了 5 个信号给父进程。带着困惑，他来向你求助。你能解释一下这个程序有什么错误吗？

code/ecf/counterprob.c

```
1   #include "csapp.h"
2
3   int counter = 0;
4
5   void handler(int sig)
6   {
7       counter++;
8       sleep(1); /* do some work in the handler */
9       return;
10  }
11
12  int main()
13  {
14      int i;
15
16      Signal(SIGUSR2, handler);
17
18      if (Fork() == 0) { /* child */
19          for (i = 0; i < 5; i++) {
20              Kill(getppid(), SIGUSR2);
21              printf("sent SIGUSR2 to parent\n");
22          }
23          exit(0);
24      }
25
26      Wait(NULL);
27      printf("counter=%d\n", counter);
28      exit(0);
29  }
```

code/ecf/counterprob.c

图 8.36 图 8.18 中引用的技术程序

8.19 ◆◆◆

编写 fgets 函数的一个版本，叫做 tfgets，它 5 秒钟后会超时。tfgets 函数接收和 fgets 相同的输入。如果用户在 5 秒内不键入一个输入行，tfgets 返回 NULL。否则，它返回一个指向输入行的指针。

8.20 ◆◆◆◆

以图 8.20 中的示例作为开始点，编写一个支持作业控制的 shell 程序。你的 shell 必须具有以下特性：

- 用户输入的命令行由一个 name、零个或者多个参数组成，所有的都是由一个或者多个空格分隔开的。如果 name 是一个内置命令，那么 shell 就立即处理它，并等待下一个命令行。否则，shell 就假设 name 是一个可执行的文件，在一个初始的子进程（作业）的上下文中加载并运行它。作业的进程组 ID 与子进程的 PID 相同。
- 每个作业是由一个进程 ID（PID）或者一个作业 ID（JID）来标识的，它是由一个 shell 分配的小的任意正整数。JID 在命令行上用前缀“%”来表示。比如，“%5”表示 JID 5，而“5”表示 PID 5。
- 如果命令行以&来结束，那么 shell 就在后台运行这个作业。否则，shell 在前台运行这个作业。
- 输入 ctrl-c（ctrl-z），使得 shell 发送一个 SIGINT（SIGTSTP）信号给前台进程组中的每个进程。
- 内置命令 jobs 列出所有的后台作业。
- 内置命令 bg <job>通过发送一个 SIGCONT 信号重启<job>，然后在后台运行它。<job>参数可以是一个 PID，也可以是一个 JID。
- 内置命令 fg <job>通过发送一个 SIGCONT 信号重启<job>，然后在前台运行它。
- shell 回收它所有的僵死子进程。如果任何作业因为它收到一个未捕捉的信号而终止，那么 shell 就输出一条信息到终端，包含该作业的 PID 和对违规信号的描述。

图 8.37 展示了一个示例的 shell 会话。

```
unix> ./shell                                  Run your shell program
> bogus
bogus: Command not found.                      Execve can't find executable User types ctrl-c
> foo 10
Job 5035 terminated by signal: Interrupt
> foo 100 &
[1] 5036 foo 100 &
> foo 200 &
[2] 5037 foo 200 &
> jobs
[1] 5036 Running    foo 100 &
[2] 5037 Running    foo 200 &
> fg %1
Job [1] 5036 stopped by signal: Stopped        User types ctrl-z
> jobs
[1] 5036 Stopped    foo 100 &
[2] 5037 Running    foo 200 &
> bg 5035
5035: No such process
> bg 5036
[1] 5036 foo 100 &
> /bin/kill 5036
Job 5036 terminated by signal: Terminated
```

```
> fg %2                                    Wait for fg job to finish.
> quit
unix>                                      Back to the Unix shell
```

图 8.37 习题 8.20 的 shell 交互示例

练习题答案

练习题 8.1 答案

在我们图 8.13 中的示例程序中，父子进程执行无关的指令集合。然而，在这个程序中，父子进程执行的指令集合是相关的，这是有可能的，因为父子进程有相同的代码段。这会是一个概念上的障碍，所以请确认你理解了本题的答案。

A．子进程的输出是什么？这里的关键点是子进程执行了两个 printf 语句。在 fork 返回之后，它执行了第 8 行的 printf。然后它从 if 语句中出来，执行了第 9 行的 printf 语句。下面是子进程产生的输出：

```
printf1: x=2
printf2: x=1
```

B．父进程的输出是什么呢？父进程只执行了第 9 行的 printf：

```
printf2: x=0
```

练习题 8.2 答案

这个程序和图 8.14（c）中的程序有相同的进程图。一共有四个进程，每个打印一个"hello"行。因此，程序打印四个"hello"行。

练习题 8.3 答案

这个程序和图 8.14（c）有相同的进程图。一共有四个进程，每个输出一个单独的"hello"行在 doit 中，并且在它从 doit 返回后也在 main 中输出一个"hello"行。因此，这个程序就一共有八个"hello"行输出。

练习题 8.4 答案

A. 每次我们运行这个程序，就会产生六个输出行。

B. 输出行的顺序根据系统不同而不同，取决于内核如何交替执行父子进程的指令。一般而言，下图的任意拓扑排序都是有效的顺序：

```
             --> ''0'' --> ''2'' --> ''Bye''        父进程
            /
  ''Hello''
            \
             --> ''1'' --> ''Bye''                  子进程
```

比如，当我们在系统上运行这个程序时，会得到下面的输出：

```
unix> ./waitprob1
Hello
```

```
0
1
Bye
2
Bye
```

在这种情况中，父进程首先运行，在第8行打印“Hello”，在第10行打印“0”。对wait的调用会阻塞，因为子进程还没有终止，所以内核执行一个上下文切换，并将控制传递给子进程，子进程在第10行打印“1”，在第16行打印“Bye”，然后在第17行终止，退出状态为2。在子进程终止后，父进程继续，在第14行打印子进程的退出状态，在第16行打印“Bye”。

练习题8.5答案

code/ecf/snooze.c

```
1   unsigned int snooze(unsigned int secs) {
2       unsigned int rc = sleep(secs);
3       printf("Slept for %u of %u secs.\n", secs - rc, secs);
4       return rc;
5   }
```

code/ecf/snooze.c

练习题8.6答案

code/ecf/myecho.c

```
1   #include "csapp.h"
2
3   int main(int argc, char *argv[], char *envp[])
4   {
5       int i;
6
7       printf("Command line arguments:\n");
8       for (i=0; argv[i] != NULL; i++)
9           printf("    argv[%2d]: %s\n", i, argv[i]);
10
11      printf("\n");
12      printf("Environment variables:\n");
13      for (i=0; envp[i] != NULL; i++)
14          printf("    envp[%2d]: %s\n", i, envp[i]);
15
16      exit(0);
17  }
```

code/ecf/myecho.c

练习题8.7答案

只要休眠进程收到一个未被忽略的信号，sleep函数就会提前返回。但是，因为收到一个SIGINT信号的默认行为就是终止进程（图8.23），我们必须设置一个SIGINT处理程序来允许sleep函数返

回。处理程序简单地捕捉 SIGNAL，并将控制返回给 sleep 函数，该函数会立即返回。

code/ecf/snooze.c

```
1   #include "csapp.h"
2
3   /* SIGINT handler */
4   void handler(int sig)
5   {
6       return; /* catch the signal and return */
7   }
8
9   unsigned int snooze(unsigned int secs) {
10      unsigned int rc = sleep(secs);
11      printf("Slept for %u of %u secs.\n", secs - rc, secs);
12      return rc;
13  }
14
15  int main(int argc, char **argv) {
16
17      if (argc != 2) {
18          fprintf(stderr, "usage: %s <secs>\n", argv[0]);
19          exit(0);
20      }
21
22      if (signal(SIGINT, handler) == SIG_ERR) /* install SIGINT handler */
23          unix_error("signal error\n");
24      (void)snooze(atoi(argv[1]));
25      exit(0);
26  }
```

code/ecf/snooze.c

CHAPTER 9

测量程序执行时间

人们经常问的一个问题是："程序X在机器Y上运行得有多快？"一个试图优化程序性能的程序员，或者一个想要决定买哪台机器的顾客，可能会提出这样的问题。在我们前面对性能优化的讨论中（第5章），我们假设能够非常准确地回答这个问题。我们试图把程序的CPE（每元素的周期数）测量值精确到小数点后两位。对一个CPE为10的过程，这要求精确度为0.1%。在本章中，我们会讲述这个问题，并会发现它是非常复杂的。

你可能会以为在计算机系统上获得几近完美的计时测量会很简单。毕竟，对于某个程序和数据的组合，机器会执行固定的指令序列。指令的执行是由处理器时钟控制的，而处理器时钟是由精度振荡器控制的。不过，一个程序的执行与另一个程序的执行之间有许多因素是不同的。计算机并不同时只执行一个程序。它们不停地从一个进程切换到另一个，为一个进程执行一些代码，然后再移到下一个进程。对一个程序的处理器资源的准确调度依赖于这样一些因素，例如共享系统的用户数量、网络流量和对磁盘操作的计时。对高速缓存的访问模式不仅仅依赖于我们正在试图测量的程序的引用，还依赖于同时正在执行的其他进程。最后，分支预测逻辑会根据以往的历史猜测是否会选择分支。一个程序每次执行的历史都不相同。

在本章中，我们描述计算机用来记录时间流逝的两种基本机制：一种基于低频率计时器（timer），它会周期性中断处理器；另一种基于计数器（counter），每个时钟周期计数器会加1。应用程序的程序员可以通过调用库函数获得对前一种计时机制的访问。有些系统上，可以通过库函数访问周期计时器（cycle timer），但是有些系统上需要编写汇编代码。我们将程序计时推迟到现在才讨论，是因为程序计时需要理解CPU硬件和操作系统管理进程执行的方式。

使用这两种计时机制，我们来研究获得程序性能的可靠测量值的方法。我们会看到，由于上下文切换引起的计时变化会非常大，因此必须消除。由其他因素引起的变化，例如高速缓存和分支预测，通常是通过在精心控制的条件下执行程序操作来管理的。一般来说，我们可以获得对于非常短（小于大约10ms）或者非常长（大于大约1s）的时间段的准确测量值，即使是在负载很重的机器上。10ms～1s之间的时间要想准确测量需要特殊的处理。

许多对性能测量的理解都是计算机系统传说的一部分。不同的小组和个人开发了他们自己的测量程序性能的技术，但是关于这个主题没有广泛流传的文献。那些专业性能测量的公司和研究组，常常建立特殊配置的机器，使得造成计时不规则的来源最少，例如，通过限制访问或者关掉许多OS和网络服务。我们希望能有程序员在普通机器上就能使用的方法，但是没有这样的广泛可获得的工具。所以，我们会开发我们自己的工具。

在这里的描述中，我们会系统地讲述这些问题。我们描述大量实验的设计和评价，这些实验帮助我们获得在一规模系统上取得准确测量的方法。在一本这个层次的书中找到详细的实验研究还是不太常见的。通常，人们只想要最后的答案，而不想知道是怎样确定这些答案的。不过，在这里，对于如何在任意系统上测量任意程序的执行时间，我们不能提供确定的答案。有太多的计时机制、操作系统行为和运行时环境，不可能有一个惟一的、简单的解决方案。相反，我们期望你自己做实验，开发你自己的性能测量代码。我们希望我们的案例研究能帮助你完成这项任务。我们把我们的发现以协议的形式总结出来，它能够指导你的实验。

9.1 计算机系统上的时间流

计算机是在两个完全不同的时间尺度（time scale）上工作的。在微观级别，它们以每个时钟周期一条或多条指令的速度执行指令，这里每个时钟周期只需要大约 1ns（纳秒）或者 10^{-9}s。在宏观尺度上，处理器必须响应外部事件，外部事件发生的时间尺度要以 ms（毫秒）或者 10^{-3}s 来度量。例如，在视频播放时，大多数计算机的图形显示器必须每 33ms 刷新一次。保持世界记录的打字员敲键盘的速度也只能是大约每 50ms 一次击键。磁盘通常需要大约 10ms 来启动一次磁盘传送。在宏观时间尺度上，处理器不停地在许多任务之间切换，一次分配给每个任务大约 5～20ms。以这样的速度，用户感觉上任务是在同时进行的，因为人不能够察觉短于大约 100ms 的时间段。在这段时间内，处理器可以执行几百万条指令。

图 9.1 在对数尺度上画出了各种事件类型的持续时间，微观事件的持续时间以 ns 为单位，（大）宏观事件的持续时间以 ms 为单位。（小）宏观事件是由 OS 例程来管理的，需要大约 5 000～200 000 个时钟周期。这些时间范围是以μs 来测量的（微秒，这里μ是希腊字符 “mu”）。听上去好像是很多的计算，但是它比处理（大）宏观事件要快很多，以至于这些例程只给处理器增加了少量的负载。[1]

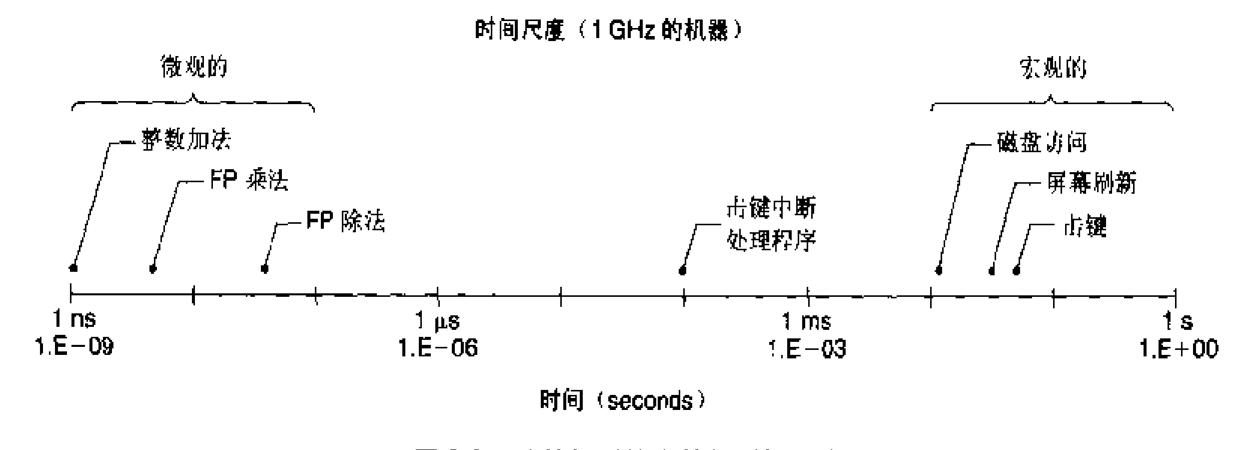


图 9.1 计算机系统事件的时间尺度

处理器硬件在微观时间尺度上工作，在这个级别上事件的持续时间都是几 ns 量级的。OS 必须以宏观时间尺度来处理持续时间为几 ms 量级上的事件。

练习题 9.1

当用户用 EMACS 这样的实时编辑器来编辑文件时，每次击键都产生一个中断信号。然后，操作系统必须调度编辑器进程，对这次击键采取适当的行动。假设我们有一个时钟为 1 GHz 的系统，而我们有 100 个用户在运行 EMACS，他们以每分钟 100 个单词的速度输入。假设每个单词平均有 6 个字符。还假设处理击键的 OS 例程平均需要 100 000 个时钟周期/键。对所有这些击键的处理占用了处理器负载的百分之多少？

注意，这是对键盘使用造成的负载非常悲观的分析。很难想像现实生活中有这么多输入如此快的用户。

1 原文并没有区分（大）宏观级别和（小）宏观级别，这样理解本段将很困难。——译者

9.1.1 进程调度和计时器中断

外部事件，例如击键、磁盘操作和网络活动，会产生中断信号，这些中断信号使得操作系统调度程序得以运行，可能还会切换到另一个进程。即使没有这样的事件，我们也希望处理器从一个进程切换到另一个，这样用户看上去就好像处理器在同时执行许多程序一样。出于这个原因，计算机有一个外部计时器，它周期性地向处理器发送中断信号。这些中断信号之间的时间被称为间隔时间（interval time）。当计时器中断发生时，操作系统调度程序可以选择要么继续当前正在执行的进程，要么切换到另一个进程。这个间隔必须设置得足够短，以保证处理器在任务间切换得足够频繁，能够提供在同时执行多个任务的假象。另一方面，从一个进程切换到另一个进程需要几千个时钟周期来保存当前进程的状态，并且为下一个进程准备好状态，因此将间隔设置得太短会导致性能很差。根据处理器以及处理器的配置情况，典型的计时器间隔范围是1～10ms。

旁注：计算机性能的伸缩

将 Digital Equipment Corporation 的 VAX-11/780 计算机的性能比喻成一个现代处理器，这是很有意思的。这种机器是在 1977 年出现的，每台售价大约是 200 000 美元。它成为第一种被广泛使用的运行 Unix 操作系统的机器。注意，这种机器上的计时器间隔典型地被设置为 10ms，即使它的 CPU 比现代机器的 CPU 慢了 1 000 倍。虽然微观时间尺度变化得飞快，但是宏观时间尺度并没有改变太多。

图 9.2（a）从系统的角度说明了在计时器间隔为 10ms 的系统上一个假设的 150ms 的操作。在这段时间内有两个活动的进程：A 和 B。处理器交替地执行进程 A 的一部分，然后再执行 B 的一部分，依此类推。当处理器执行这些进程时，它要么运行在用户模式，执行应用程序的指令，要么运行在内核模式，代表程序执行操作系统函数，例如处理缺页、输入或者输出。回想一下，内核操作被认为是每个普通进程的一部分，而不是一个独立的进程。每次有外部事件或者计时器中断时，都会调用操作系统调度程序。在图中，计时器中断的发生是由短线标记来表示的。这意味着在每个短线标记处都有一些内核活动，但是为了简便，在图中我们没有显示。

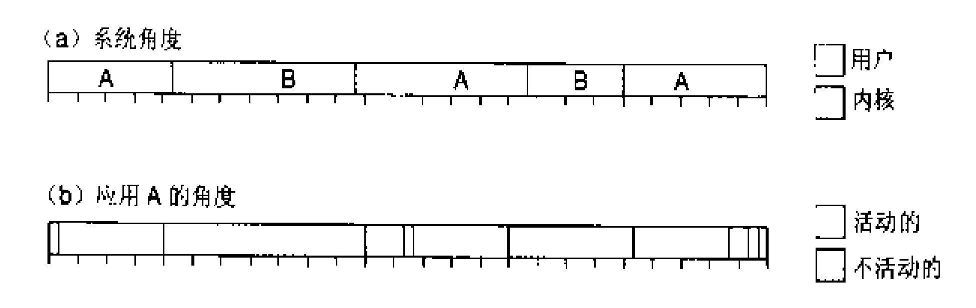


图 9.2 系统和应用对时间的看法

系统从一个进程切换到另一个进程，这些进程运行在用户模式或者内核模式。当应用的进程在用户模式中执行时，应用才能完成有用的计算。

当调度程序从进程 A 切换到进程 B 时，它必须进入内核模式保存进程 A 的状态（仍被认为是进程 A 的一部分），然后恢复进程 B 的状态（被认为是进程 B 的一部分）。因此，在每次从一个进程过渡到另一个进程期间，是有内核活动的。在其他时候，也有除了切换进程之外的内核活动，例如当通过使用一个已经在存储器（memory）中的页来满足缺页时。

9.1.2 从应用程序的角度看时间

从应用程序的角度出发，可以把时间流看成两种时间段的交替，一种时间段里程序是*活动的*（在执行它的指令），另一种时间段里程序是*不活动的*（等待被操作系统调度）。当应用的进程运行在用户模式中时，应用才能执行有用的计算。图 9.2（b）说明了程序 A 是如何看待时间流的。在深灰色区域内应用是活动的，此时进程 A 正在用户模式中执行；否则它是不活动的。

作为一种量化在活动和不活动时间段之间交替的方法，我们写了一个程序[2]，它不断地监视它自己，确定什么时候有长的不活动时间。然后它产生一个 trace（跟踪文件），显示出在活动和不活动时间段之间的交替，本章后面会描述这个程序的细节。图 9.3 展示了一个这样的 trace 示例，是在一个时钟周期大约为 550 MHz 的 Linux 机器上运行时产生的。每个时间段都标记为活动的（“A”）或者不活动的（“I”）。时间段被编号为 0～9，用来标识。对于每个时间段，给出了开始时间（相对于 trace 的开始）和持续时长。以时钟周期和 ms 来表示时间。这个 trace 一共显示了 20 个时间段（10 个活动的，10 个不活动的），总共的持续时间是 66.9 ms。在这个例子中，不活动时间段相当短，最长的是 0.50 ms。大多数不活动时间段是由计时器中断造成的。被监视的总时间中，大约 95.1%的时间这个进程都是活动的。图 9.4 展示了图 9.3 所示的 trace 的图形化表示。注意，灰色三角形指明了活动时间段之间边界的规则间隔。这些边界是由计时器中断造成的。

```
A0  时间         0   (0.00 ms),   持续时间  3726508    (6.776448 ms)
I0  时间   3726508   (6.78 ms),   持续时间   275025    (0.500118 ms)
A1  时间   4001533   (7.28 ms),   持续时间        0    (0.000000 ms)
I1  时间   4001533   (7.28 ms),   持续时间     7598    (0.013817 ms)
A2  时间   4009131   (7.29 ms),   持续时间  5189247    (9.436358 ms)
I2  时间   9198378   (16.73 ms),  持续时间   251609    (0.457537 ms)
A3  时间   9449987   (17.18 ms),  持续时间  2250102    (4.091686 ms)
I3  时间  11700089   (21.28 ms),  持续时间    14116    (0.025669 ms)
A4  时间  11714205   (21.30 ms),  持续时间  2955974    (5.375275 ms)
I4  时间  14670179   (26.68 ms),  持续时间   248500    (0.451883 ms)
A5  时间  14918679   (27.13 ms),  持续时间  5223342    (9.498358 ms)
I5  时间  20142021   (36.63 ms),  持续时间   247113    (0.449361 ms)
A6  时间  20389134   (37.08 ms),  持续时间  5224777    (9.500967 ms)
I6  时间  25613911   (46.58 ms),  持续时间   254340    (0.462503 ms)
A7  时间  25868251   (47.04 ms),  持续时间  3678102    (6.688425 ms)
I7  时间  29546353   (53.73 ms),  持续时间     8139    (0.014800 ms)
A8  时间  29554492   (53.74 ms),  持续时间  1531187    (2.784379 ms)
I8  时间  31085679   (56.53 ms),  持续时间   248360    (0.451629 ms)
A9  时间  31334039   (56.98 ms),  持续时间  5223581    (9.498792 ms)
I9  时间  36557620   (66.48 ms),  持续时间   247395    (0.449874 ms)
```

图 9.3 显示活动时间段的示例 trace

从应用程序的角度来看，处理器操作是在程序活动执行（以斜体表示的）和不活动之间交替进行的。这个 trace 展示了一个程序在 66.9 ms 时间段内两种时间段的日志记录。有 95.1%的时间程序是活动的。

图 9.5 展示了一个 trace 的一部分，此时还有另一个活动进程在共享处理器。图 9.6 中展示了这

2 即下文提到的跟踪进程。——译者

个 trace 的图形化表示。注意，两幅图中的时间尺度不一样，因为我们显示的这部分 trace 是从跟踪进程的 349.40 ms 处开始的。在这个例子中，我们可以看到，在处理某些计时器中断时，OS 也会决定从一个进程切换上下文到另一个进程。因此，每个进程只会在大约 50%的时间里是活动的。

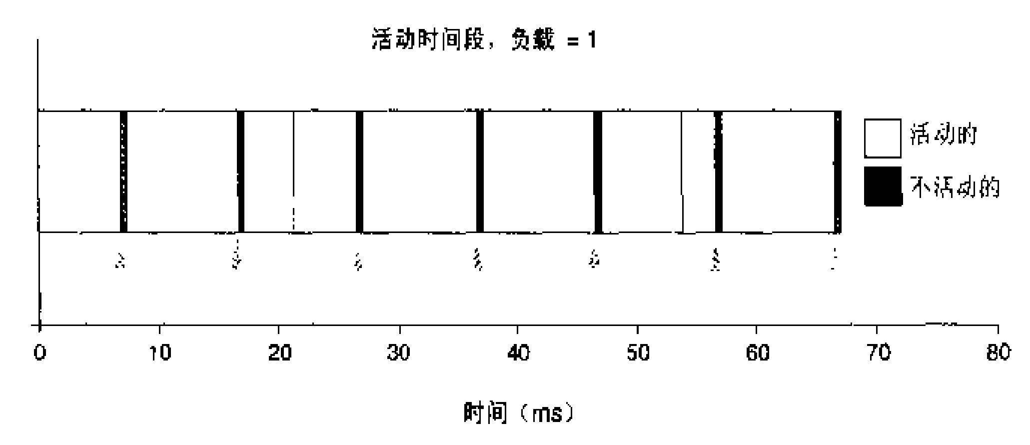


图 9.4 图 9.3 中 trace 的图形化表示

计时器中断是由灰色三角形来指示的。

```
A48  时间  191514104  (349.40 ms),  持续时间  5224961  (9.532449 ms)
I48  时间  196739065  (358.93 ms),  持续时间   247557  (0.451644 ms)
A49  时间  196986622  (359.38 ms),  持续时间   858571  (1.566382 ms)
I49  时间  197845193  (360.95 ms),  持续时间     8297  (0.015137 ms)
A50  时间  197853490  (360.97 ms),  持续时间  4357437  (7.949733 ms)
I50  时间  202210927  (368.91 ms),  持续时间  5718758  (10.433335 ms)
A51  时间  207929685  (379.35 ms),  持续时间  2047118  (3.734774 ms)
I51  时间  209976803  (383.08 ms),  持续时间     7153  (0.013050 ms)
A52  时间  209983956  (383.10 ms),  持续时间  3170650  (5.784552 ms)
I52  时间  213154606  (388.88 ms),  持续时间  5726129  (10.446783 ms)
A53  时间  218880735  (399.33 ms),  持续时间  5217543  (9.518916 ms)
I53  时间  224098278  (408.85 ms),  持续时间  5718135  (10.432199 ms)
A54  时间  229816413  (419.28 ms),  持续时间  2359281  (4.304286 ms)
I54  时间  232175694  (423.58 ms),  持续时间     7096  (0.012946 ms)
A55  时间  232182790  (423.60 ms),  持续时间  2859227  (5.216390 ms)
I55  时间  235042017  (428.81 ms),  持续时间  5718793  (10.433399 ms)
```

图 9.5 显示有负载机器上的活动时间段的示例 trace

当还有其他活动进程存在时，跟踪进程会较长时间不活动。这个 trace 展示了一个程序在总长为 89.8ms 的时间段内的日志。跟踪进程在 53.0%的时间内都是活动的。

练习题 9.2

这个问题是关于图 9.5 所示 trace 的一部分的解释的。

A. 在这部分 trace 中，什么时候发生了计时器中断？（其中有些时间点能够直接从 trace 中提取出来，而有些必须用插值法估计。）

B. 这些计时器中断中，哪些是在跟踪进程活动时发生的，哪些是在它不活动时发生的？

C. 为什么最长的不活动时间段比最长的活动时间段要长呢？

D. 根据这个 trace 中活动和不活动时间段的模式，你预计在更长的时间范围内，跟踪进程处于不活动状态的时间百分比会是多少？

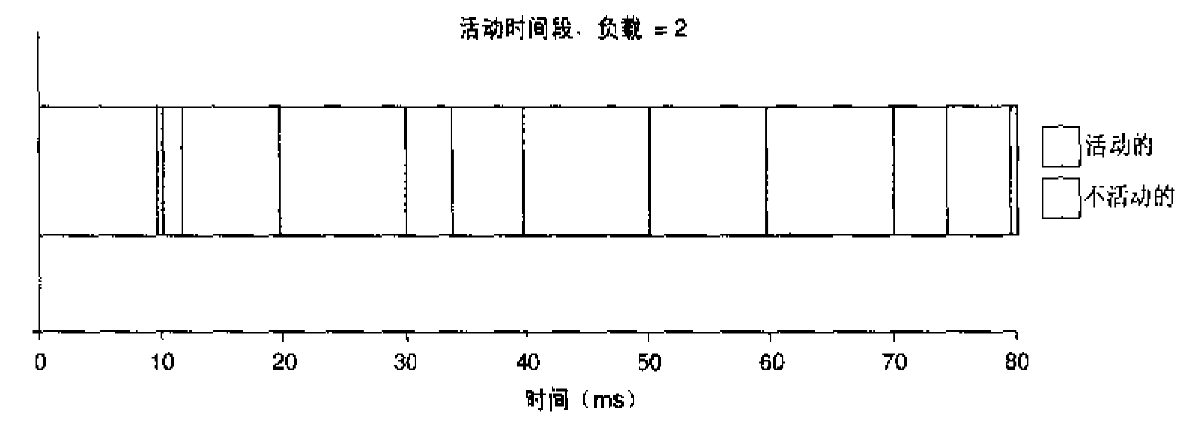


图 9.6　图 9.5 中 trace 的活动时间段的图形化表示

计时器中断是由灰色三角形来指示的。

9.2　通过间隔计数（interval counting）来测量时间

操作系统也用计时器（timer）来记录每个进程使用的累计时间，这种信息提供的是对程序执行时间不那么准确的测量值。图 9.7 提供了如何对图 9.2 中所示的系统操作示例进行这种记账（accounting）的图形化说明。在这里的讨论中，我们称只有一个进程在执行的一段时间为时间段（time segment）[3]。

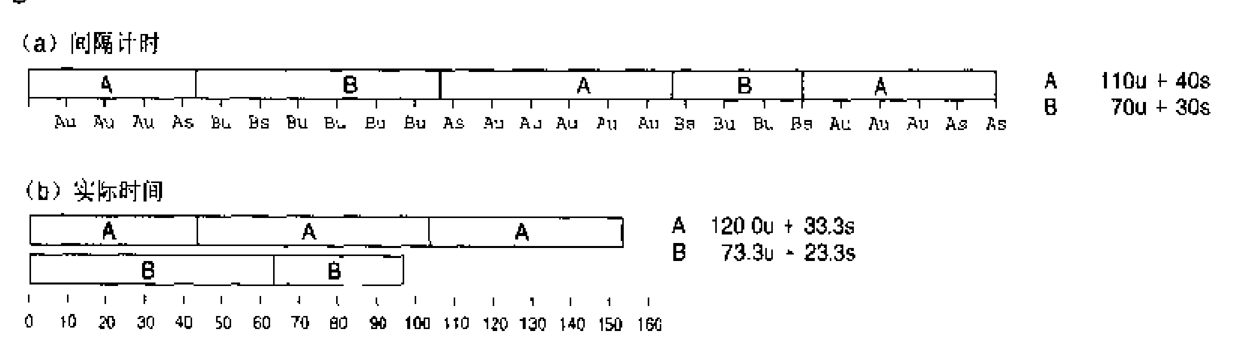


图 9.7　通过间隔计数（interval counting）来对进程计时

计时器间隔为 10ms 时，每 10ms 时间段被分配给一个进程，作为它的用户时间（u）或者系统时间（s）的一部分。这样的记账（accounting）提供的只是程序执行时间的一个粗略的测量值。

9.2.1　操作

操作系统维护着每个进程使用的用户时间量和系统时间量的计数值，当计时器中断发生时，操作系统会确定哪个进程是活动的，并且对那个进程的一个计数值增加计时器间隔时间。如果系统是在内核模式中执行的，那么就增加系统时间，否则就增加用户时间。图 9.7（a）所示的例子表明了对两个进程的这种记账（accounting）。短线标记表明发生了计时器中断。每个计时器中断都由被增加的计数值来标识：或者是进程 A 的用户或系统时间 Au 或 As，或者是进程 B 的用户或系统时间

3　通常称为时间片。——译者

Bu 或 Bs。每个短线标记是根据紧挨着它的左边的活动来标识的。最后的记账表明进程 A 总共使用了 150ms：110ms 的用户时间和 40ms 的系统时间；进程 B 总共使用了 100ms：70ms 的用户时间和 30ms 的系统时间。

9.2.2 读进程的计时器

当从 Unix shell 执行一个命令时，用户可以在命令前加上单词“time”，来测量命令的执行时间。这个命令使用的值是用上面描述的记账方法计算出来的。例如，为了计算命令行参数为 -n 17 的程序 prog 的执行时间，用户只要简单地输入命令：

```
unix> time prog -n 17
```

在程序执行完毕之后，shell 会打印出总结运行时间数据的一行，就像下面的这样：

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

这一行中显示的头三个数字是时间。前两个是用户和系统时间的秒数。注意这两个数字的小数点后第三位都是 0。计时器间隔为 10 ms，所有的计时都是百分之一秒的倍数。第三个数字是总共经过的时间，以分钟和秒的形式表示。我们注意到系统和用户时间加起来是 2.49s，比总共经过的时间 6.52s 的一半还要少，这表明处理器同时还在执行其他的进程。百分比表明用户和系统时间的和占经过时间的比例，例如，(2.23 + 0.26)/6.52 = 0.381。剩下的统计数据总结了页面调度和 I/O 行为。

程序员还可以通过调用库函数 times 来读进程的计时器，这个函数的声明如下：

```
#include <sys/times.h>

struct tms {
    clock_t tms_utime;    /* user time */
    clock_t tms_stime;    /* system time */
    clock_t tms_cutime;   /* user time of reaped children */
    clock_t tms_cstime;   /* system time of reaped children */
};
clock_t times(struct tms *buf);
```

返回：自系统启动以来经过的时钟滴答数。

这些时间测量值是以时钟滴答（clock tick）为单位来表示的。定义的常数 CLK_TCK 指明每秒的时钟滴答数。数据类型 clock_t 通常定义为长整型。指明子时间的字段给出的是已经终止了并且被回收了的子进程使用的累积时间。因此，times 不能用来监视任何正在进行的子进程所使用的时间。作为返回值，times 返回的是从系统启动开始已经经过的时钟滴答总数。因此我们可以通过两次调用 times，再计算两个返回值的差，来计算一个程序执行中两个不同点之间的总时间（以时钟滴答为单位）。

ANSI C 标准还定义了一个 clock 函数，它测量当前进程使用的总时间：

```
#include <time.h>

clock_t clock(void);
```

返回：进程使用的总时间。

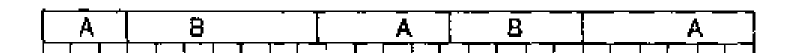
虽然它的返回值和函数 times 使用的一样，都声明为 clock_t 类型，但是通常这两个函数表达时间的单位是不一样的。要将 clock 函数报告的时间变成秒数，必须把它除以定义好的常数 CLOCKS_PER_SEC。这个常数值不一定要和常数 CLK_TCK 相同。

9.2.3 进程计时器的准确性

如图 9.7 所示的示例，这个计时机制只是近似的。图 9.7（b）展示了两个进程实际使用的时间。进程 A 总共执行了 153.3ms，其中用户模式 120.0 ms，内核模式 33.3ms。进程 B 总共执行了 96.7ms，其中用户模式 73.3ms，内核模式 23.3ms。采用间隔计数的记账（interval accounting）方法并不会比计时器间隔（timer interval）方法更好地解决时间问题。

练习题 9.3

操作系统会怎样报告下面所示执行序列的用户和系统时间？假设计时器间隔为 10 ms。



练习题 9.4

在一个计时器间隔为 10 ms 的系统上，进程 A 的某一个时间段被记录为需要 70 ms，包括系统和用户时间。这个片段使用的最大和最小实际时间是多少？

练习题 9.5

对于图 9.3 中所示的 trace，间隔计数器（counter）记录的系统和用户时间会是多少？这个时间与进程处于活动状态的实际时间之比为多少？

对于运行时间足够长的程序（至少要几秒钟），这种方法中的不准确性就能相互弥补了。一些时间段的执行时间被低估了，而另一些被高估了。在许多时间段上一平均，期望的误差就接近于 0 了。不过，从理论的角度来看，对于这种测量值与真实运行时间的差距有多大，并没有确切的界限。

为了测试这种计时方法的准确性，我们运行了一系列实验，比较相同样本计算下操作系统所测量的值 T_m 和如果系统资源只用来执行这个计算时我们估计的时间 T_c。一般而言，T_c 与 T_m 不相同有以下几个原因：

1. 间隔计数方法本身固有的不准确性可以导致 T_m 比 T_c 小或者大。

2. 计时器中断导致的内核活动占用了总 CPU 周期的 4%~5%，但是对这些周期的计数不是很适当。正如从图 9.4 所示的 trace 中可以看到的那样，这个活动在下一次计时器中断之前结束，因此没有显式地被算进去。相反，它只简单地减少了下一个时间间隔内执行进程的可用周期数。相对于 T_c，这就增加了 T_m。

3. 当处理器从一个任务切换到另一个任务时，在一个短暂的时间内，高速缓存可能会执行效率很差，直到新任务的指令和数据被加载到高速缓存中。因此，当处理器在我们的程序与其他活动之间切换时，它的执行效率没有连续执行我们程序时的效率高。相对于 T_c，这个因素会增加 T_m。

在本章后面，我们将讨论如何确定我们示例计算的 T_c 值。

图 9.8 给出了在两种不同的负载条件下运行这个试验的结果。这些曲线图展示了我们的误差率的测量值，它定义为 T_c 的一个函数 $(T_m - T_c)/T_c$ 的值。当 T_m 估计的低于 T_c 时，这个误差测量值为负，当 T_m 估计的高于 T_c 时，它为正。两组数据显示的是在两种不同负载条件下测量的值。标号为“负

载1”的那组数据显示的是执行示例计算的进程是惟一活动进程时的情况。标号为“负载11”的那组数据显示的是另外还有10个进程也在试图进行同样的计算时的情况。后者代表的是一个负载非常重的情况，系统对击键和其他服务请求的响应明显慢了。注意，这幅图中显示的误差值范围很大。一般而言，只有在真实值±10%范围内的测量值才是可接受的，因此我们只希望误差变化范围为大约-0.1～+0.1。

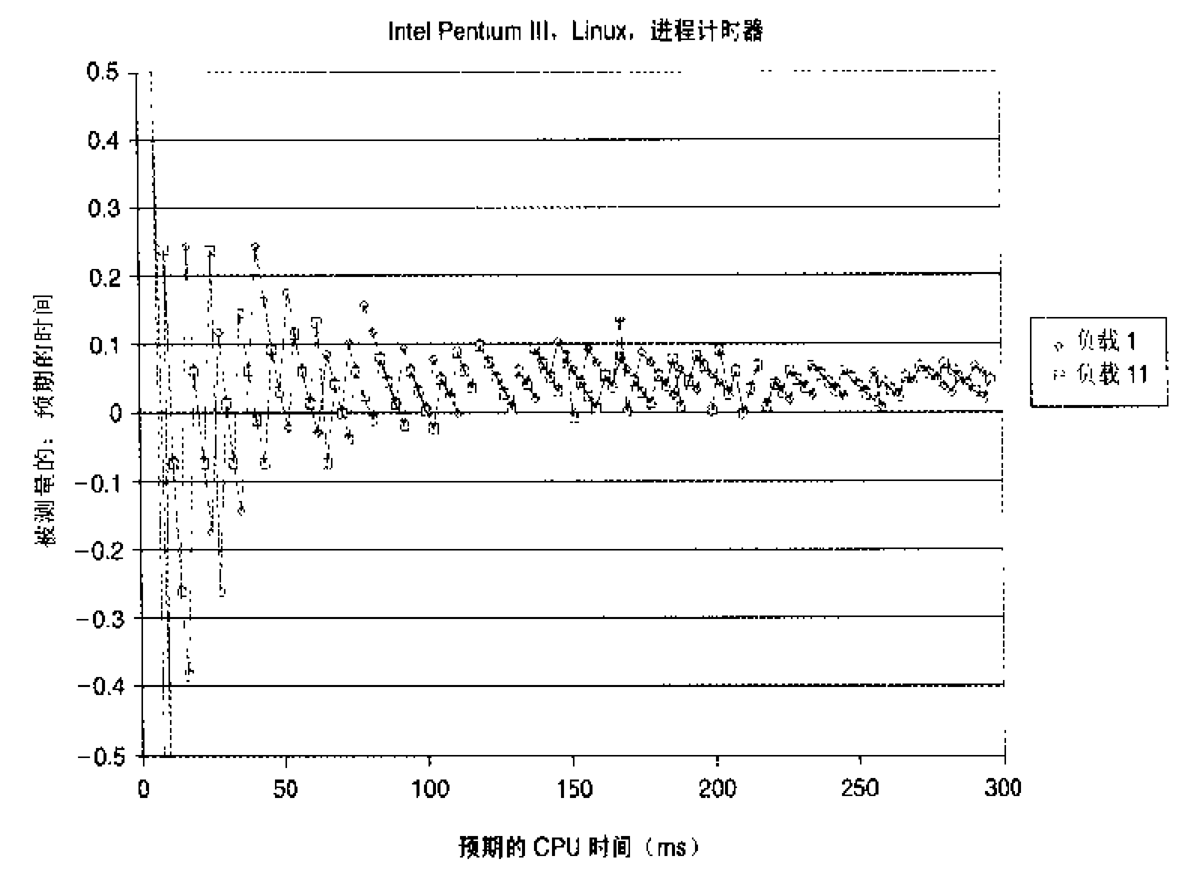


图 9.8 测量间隔计数（interval counting）的准确性

当测量活动短于大约100 ms（10个计时器间隔）时，误差会不可接受得高。此外，无论是运行在负载非常轻（负载1）的机器上，还是在负载非常重的机器上（负载11），误差率通常都小于10%。

低于大约100ms（10个计时器间隔），由于计时方法很粗糙，测量完全不准确。间隔计数只对测量相对较长的计算——100 000 000个时钟周期或更多——有用。除此之外，我们还看到误差通常在0.0～0.1之间，也就是，最多有10%的误差。两种不同的负载情况之间没有明显的区别。另外还要注意，误差有正偏差：对于所有 $T_m \geq 100$ ms 的测量值，平均误差为1.04，这是因为计时器中断占用了大约4%的CPU时间。

这些实验表明进程计时器只对获得程序性能的近似值有用。它们的粒度太粗，不能用于持续时间小于100ms的测量。在这台机器上，这些进程计时器有系统偏差，过高地估计计算时间，平均大约4%。这种计时机制的主要优点是它的准确性不是非常依赖于系统负载。

9.3 周期计数器

为了给计时测量提供更高的精确度，许多处理器还包含一个运行在时钟周期级的计时器。这个

计时器是一个特殊的寄存器，每个时钟周期它都会加 1。可以用特殊的机器指令来读这个计数器的值。不是所有的处理器都有这样的计数器的，而且有这样计数器的处理器在实现细节上也各不相同。因此，程序员无法用统一的、与平台无关的接口使用这些计数器。另一方面，只用少量的汇编代码，通常很容易就为某个特定的机器创建一个程序接口。

9.3.1 IA32 周期计数器

到目前为止，我们已经报告的所有计时值都是用 IA32 周期计数器（cycle counter）测量出来的。在 IA32 体系结构中，周期计数器是与“P6”微体系结构（PentiumPro 及其后续产品）一起提出来的。周期计数器是一个 64 位无符号数。对于一个运行时钟为 1 GHz 的处理器，只有在每 1.8×10^{10} 秒，或者每 570 年，这个计数器才会从 $2^{64}-1$ 绕回到 0。另一方面，如果我们只考虑这个计数器的低 32 位，把它看成一个无符号整数，那么这个值会大约每 4.3 秒就绕回来。因此，我们就明白了为什么 IA32 的设计者会决定实现一个 64 位的计数器。

IA32 计数器是用 rdtsc（read time stamp counter，读时间戳计数器）指令来访问的。这条指令没有参数。它将寄存器%edx 设置为计数器的高 32 位，而寄存器%eax 设置为低 32 位。为了提供一个 C 程序接口，我们想把这个指令包装到一个过程中：

```
void access_counter(unsigned *hi, unsigned *lo);
```

这个过程应该将位置 hi 设置成计数器的高 32 位，将 lo 设置成低 32 位。使用 3.15 节中描述的 GCC 的嵌入汇编特性，实现 access_counter 很简单。其代码如图 9.9 所示。

code/perf/clock.c

```
1   /* Initialize the cycle counter */
2   static unsigned cyc_hi = 0;
3   static unsigned cyc_lo = 0;
4
5
6   /* Set *hi and *lo to the high and low order bits of the cycle counter.
7      Implementation requires assembly code to use the rdtsc instruction. */
8   void access_counter(unsigned *hi, unsigned *lo)
9   {
10      asm("rdtsc; movl %%edx,%0; movl %%eax,%1"   /* Read cycle counter */
11          : "=r" (*hi), "=r" (*lo)                 /* and move results to */
12          : /* No input */                         /* the two outputs */
13          : "%edx", "%eax");
14  }
15
16  /* Record the current value of the cycle counter. */
17  void start_counter()
18  {
19     access_counter(&cyc_hi, &cyc_lo);
20  }
21
```

```
22  /* Return the number of cycles since the last call to start_counter. */
23  double get_counter()
24  {
25      unsigned ncyc_hi, ncyc_lo;
26      unsigned hi, lo, borrow;
27      double result;
28
29      /* Get cycle counter */
30      access_counter(&ncyc_hi, &ncyc_lo);
31
32      /* Do double precision subtraction */
33      lo = ncyc_lo - cyc_lo;
34      borrow = lo > ncyc_lo;
35      hi = ncyc_hi - cyc_hi - borrow;
36      result = (double) hi * (1 << 30) * 4 + lo;
37      if (result < 0) {
38          fprintf(stderr, "Error: counter returns neg value: %.0f\n", result);
39      }
40      return result;
41  }
```

code/perf/clock.c

图 9.9　实现 IA32 周期计数器的程序接口的代码

需要汇编代码来使用计数器读指令。

基于这个例程，现在我们能够实现两个函数，可以用它们来测量任意两个时间点之间经过的时钟周期总数。

```
#include "clock.h"

void start_counter();

double get_counter();
```

返回：自最后一次调用启动计数器所经过的周期数。

我们返回的时间是 double 类型的，以避免只使用 32 位整数可能引起的溢出问题。这两个例程的代码也显示在图 9.9 中。它是建立在我们对执行双精度减法和将结果转换成 double 类型的无符号运算的理解的基础上的。

9.4　用周期计数器来测量程序执行时间

周期计数器（cycle counter）提供了一个非常精确的工具，可以测量一个程序执行中两个不同点之间经过的时间。不过，典型地，我们对测量执行某段特殊代码所需要的时间感兴趣。我们的周期计数器例程计算调用 start_counter 和调用 get_counter 之间总的周期数。这些例程不记录哪个进程使用这些周期，或者处理器是在内核还是在用户模式中运行的。在使用这样的测量设备来确定执行时

间时，我们必须很小心。我们研究一些其中的困难之处，并看看如何克服它们。

作为一个使用周期计数器的代码示例，图 9.10 中的例程提供了一个确定处理器时钟频率的方法。在几个系统上，以参数 sleeptime 等于 1 来测量这个函数，测量值表明它报告的时钟频率在该处理器评测性能的 1.0%范围之内。这个示例清楚地表明我们的例程测量的是经过的时间，而不是某个进程使用的时间。当我们的程序调用 sleep 时，操作系统不会继续这个进程，直到 1s 的睡眠时间到达。这期间经过的周期是被其他进程执行的。

code/perf/clock.c

```
1   /* Estimate the clock rate by measuring the cycles that elapse */
2   /* while sleeping for sleeptime seconds */
3   double mhz(int verbose, int sleeptime)
4   {
5       double rate;
6
7       start_counter();
8       sleep(sleeptime);
9       rate = get_counter() / (1e6*sleeptime);
10      if (verbose)
11          printf("Processor clock rate ~= %.1f MHz\n", rate);
12      return rate;
13  }
```

code/perf/clock.c

图 9.10 函数 mhz：确定一个处理器的时钟频率

9.4.1 上下文切换的影响

测量某个过程 P 的运行时间的一种简单方法就是用周期计数器来对 P 的一次执行进行计时，就像在下列代码中一样：

```
1   double time_P()
2   {
3       start_counter();
4       P();
5       return get_counter();
6   }
```

如果在两次调用计数器例程之间，有另外某个进程执行了，那么这段代码就很容易产生令人误解的结果。如果机器负载很重，或者如果 P 的运行时间特别长，这就特别成问题。图 9.11 说明了这一现象。图中展示了反复测量一个程序的结果，这个程序计算的是一个 131 072 个整数的数组的和。时间被转换成了以 ms 为单位。注意总的运行时间是 36 ms，比计时器间隔值大[4]。我们进行两组测量，每组对同一个过程测量 18 次。标号为“负载 1”的那组数据说明的是在负载很轻的机器上的运行时间，此时机器上只有一个进程在运行。所有的测量值都在最小运行时间的 3.4%范围之内。标号

4 操作系统根据计时器间隔的值来执行进程调度。——译者

为“负载4”的那组数据表明的是当另外还有三个频繁使用CPU和存储器系统的进程在运行时的运行时间。头七个样本的时间在负载1样本中最快的时间的2%范围之内，但是其他的时间比4.3倍还多。

正如这个示例说明的那样，上下文切换导致执行时间差异极大。如果一个进程被交换出去[5]，那么它就会落后百万条指令。显然，我们设计的任何测量程序执行时间的方法都必须避免这样大的误差。

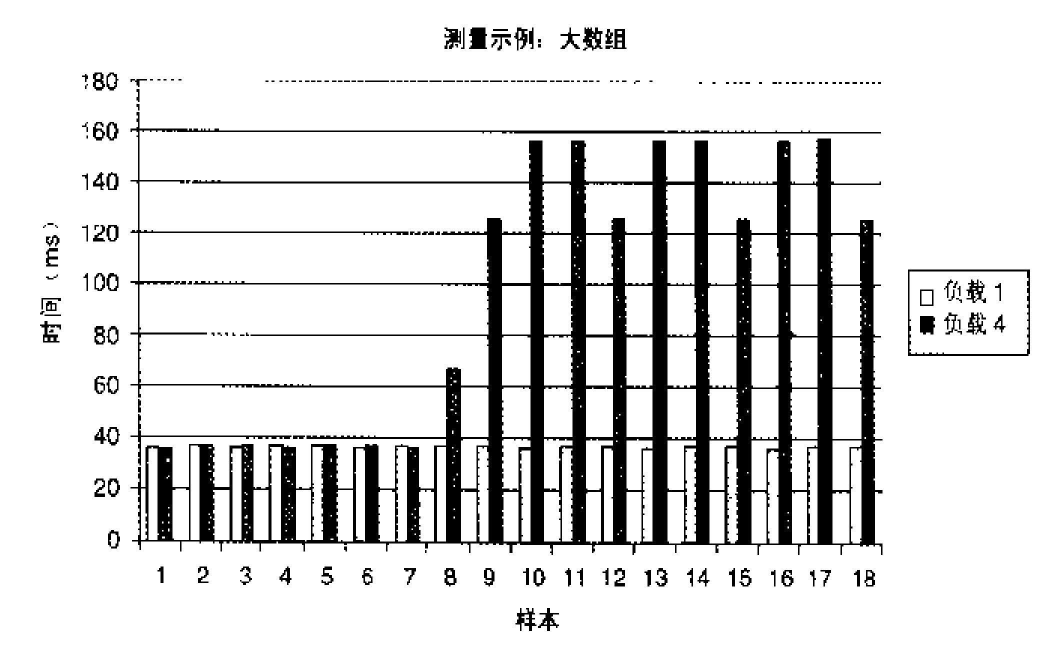


图9.11 在不同的负载情况下，对长持续时间的过程的测量

在一个负载很轻的系统上，各个样本的结果是一致的，但是在一个负载很重的系统上，许多测量值都比真实的执行时间估计得高了。

9.4.2 高速缓存和其他因素的影响

高速缓存和分支预测造成的计时变化比上下文切换造成的要小一些。作为一个例子，图9.12展示了一组类似于图9.11中的测量值，区别在于数组要小4倍，得到的执行时间大约是8ms。这些执行时间比计时器间隔要短，因此执行不太可能受上下文切换的影响。我们看到测量值有变化，但是这些变化的程度都没有上下文切换造成的变化那么大。

图9.12所示的变化主要是由高速缓存造成的。执行一个代码块的时间可以非常依赖于在开始执行时，这个代码使用的数据和指令是否在数据和指令高速缓存中。

作为一个示例，我们写了两个一样的程序procA和procB，输入为一个类型为double *的指针，并且将从这个指针开始的8个连续的元素设置为0.0。我们测量以三个不同的指针b1、b2和b3对这个过程进行调用的时钟周期数。调用序列和得到的测量值如图9.13所示。即使这些调用执行的是完全相同的计算，计时的变化也几乎有4倍。因为这段代码中没有条件分支，所以我们可以断定这些变化是受高速缓存所影响。

5 原文是time interval，而我们认为是timer interval。——译者

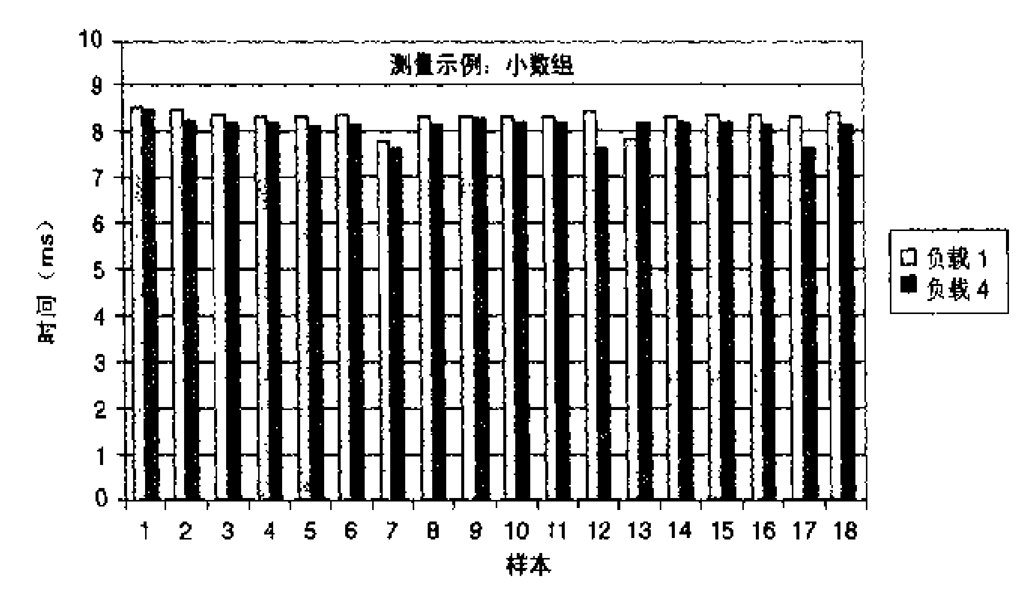


图 9.12 在不同的负载情况下，对短持续时间的过程的测量

变化程度没有图 9.11 中的变化那么大，但是还是大得不可接受。

测量	调用	周期数
1	procA(b1)	399
2	procA(b2)	132
3	procA(b3)	134
4	procA(b1)	100
5	procB(b1)	317
6	procB(b2)	100

图 9.13 相同的过程在相同的数据集上的测量序列

这些测量值中的变化主要是由指令和数据高速缓存中不同的不命中情况造成的。

练习题 9.6

c 表示如果没有高速缓存不命中，调用 procA 或者 procB 所需的周期数。对于每次计算，由于高速缓存不命中浪费的周期可以分摊到每个需要取出来放到高速缓存中的数据上：

- 实现测量代码的指令（例如 start_counter、get_counter 等等）。设这些指令所需周期数为 m。
- 实现被测量过程的指令（procA 或者 procB）。设这些指令所需周期数为 p。
- 被更新的数据位置（由 b1、b2 或 b3 指示）。设这些指令所需周期数为 d。

根据图 9.13 所示的测量值，给出 c、m、p 和 d 的估计值。

给出这些测量所示的变化，人们很自然地会问：“哪一个是对的呢？”不幸地是，对这个问题没有简单的答案。这取决于我们的代码实际使用的情况，以及我们能够获得可靠测量值的情况。一个问题是测量值每次运行都不相同。图 9.13 所示的测量表显示的只是一次测量的数据。在反复的测量中，我们看到测量 1 的范围为 317～606，而测量 5 的范围为 301～326。另外，其他四次测量每次运行的变化只有几个周期。

显然，测量 1 估计过高，因为它包括了将测量代码和数据结构加载到高速缓存中的开销。进一步来说，它的变化程度最容易大。测量 5 包括了将 procB 加载到高速缓存中的开销。它的变化程度

也容易比较大。在大多数实际应用中，同样的代码会被反复执行。因此，将代码加载到指令高速缓存中的时间相对而言不太重要。我们的示例测量有点人为的痕迹，因为指令高速缓存不命中的影响比实际应用中的要大一些。

为了测量过程P所需的时间，在过程P中指令高速缓存不命中的影响已经减小到最低了，我们可以执行下列代码：

```
1   double time_P_warm()
2   {
3       P(); /* Warm up the cache */
4       start_counter();
5       P();
6       return get_counter();
7   }
```

在开始测量之前执行一次P，会将P所用的代码放入到指令高速缓存中。

这段代码也使数据高速缓存不命中的影响降低到最小，因为第一次执行P也将P访问的数据放入到数据高速缓存中。对于过程procA和procB，time_P_warm的测量会得到100个周期。如果我们预想代码会重复地访问同样的数据，那么这就是测量的正确条件。不过对于一些应用，我们更可能是每次执行都访问新的数据。例如，一个过程将数据从存储器的一个区域拷贝到另一个区域，很可能调用时没有块被缓存。过程time_P_warm倾向于低估这样一个程序的执行时间。对于procA或者procB，它会得到100个周期，而不是当过程被应用到未缓存的数据上时测出的132～134个周期。

为了使计时代码测量一个初始时没有数据被缓存了的过程，我们可以在执行实际的测量之前，清空高速缓存中所有有用的数据。下面的过程就是为一个高速缓存大小不大于512KB的系统完成这一功能的：

code/perf/time_p.c

```
1   /* Number of bytes in the largest cache to be cleared */
2   #define CBYTES (1<<19)
3   #define CINTS (CBYTES/sizeof(int))
4
5   /* A large array to bring into cache */
6   static int dummy[CINTS];
7   volatile int sink;
8
9   /* Evict the existing blocks from the data caches */
10  void clear_cache()
11  {
12      int i;
13      int sum = 0;
14
15      for (i = 0; i < CINTS; i++)
16          dummy[i] = 3;
17      for (i = 0; i < CINTS; i++)
```

```
18          sum += dummy[i];
19      sink = sum;
20  }
```

code/perf/time_p.c

这个过程只是在一个非常大的数组 dummy 上执行一个计算，有效地从高速缓存中清除出所有其他的东西。这个代码有几个特殊的性质，用来避免常见的错误。它将值存储到 dummy 中，并且把它们读出来，这样无论高速缓存分配策略是怎样的，都会缓存这个数组。这段代码用数组的值执行一个计算，并将结果存储到一个全局整数中（声明为 volatile 就表明对这个变量的任何更新都必须被执行），这样使得聪明的优化编译器不会优化掉这部分代码。

使用这个过程，我们可以获得在 P 的指令都被缓存而数据没有被缓存的情况下 P 的一个测量值：

```
1   double time_P_cold()
2   {
3       P(); /* Warm up instruction cache */
4       clear_cache(); /* Clear data cache */
5       start_counter();
6       P();
7       return get_counter();
8   }
```

当然，这个方法也有缺点。在一个有统一 L2 高速缓存的机器上，过程 clear_cache 会导致 P 的所有指令都被清除。幸运的是，L1 指令高速缓存中的指令还会保存。过程 clear_cache 还会从高速缓存中清除出大部分运行时栈，导致过高地估计了在更加真实的条件中 P 所需要的时间。

正如这里的讨论说明的那样，高速缓存的影响为性能测量增加了特殊的困难。程序员几乎不能控制什么指令和数据会被加载到高速缓存中，而当必须加载新值时又该清除什么指令和数据。最好的情况下，我们能够设置好测量条件，通过一些清空和加载高速缓存的组合，使得测量条件与我们应用期望的条件相匹配。

正如前面提到过的，分支预测逻辑也会影响程序性能，因为当分支方向和目的都预测正确时，分支指令引起的时间处罚要小得多。这个逻辑是根据已经执行过的分支指令的历史记录来进行预测的。当系统从一个进程切换到另一个时，开始时新进程中的分支预测是根据前一个进程中执行的分支指令来进行的。不过，实际上，这些影响对程序的每次执行只会造成很小的性能变化。预测主要依赖于最近的分支，因此一个进程对另一个进程的影响非常小。

9.4.3 K 次最优测量方法

虽然我们使用周期计时器测量容易受由上下文切换、高速缓存操作和分支预测引起的误差的影响，但是一个重要的特性就是这些误差总是导致过高地估计真实的执行时间。处理器做的事情都不会人为地加速一个程序的执行。即使上下文切换和其他影响会引起测量值不一致，我们仍然可以利用这个属性来获得执行时间可靠的测量值。

假设我们重复地执行一个过程，用 time_P_warm 或者 time_P_cold 来测量周期数。我们记录 K（例如 3）次最快的时间。如果我们发现这种测量的误差 ϵ 很小（如 0.1%），那么用测量的最快值

来表示过程的真实执行时间就是合理的。作为一个示例，假设对于图 9.11 所示的那些次测量，我们设置误差为 1.0%。那么负载 1 的最快的六个测量值在这个误差范围之内，而负载 4 最快的三个测量值在这个误差范围之内。因此我们可以得出结论说运行时间分别为 35.98 ms 和 35.89 ms。对于负载 4 的情况，我们还可以看到测量值集中在 125.3 ms 附近，有六个大约为 155.8 ms，但是我们安全地丢弃了这些过高的估计值。

我们称这种方法为“K 次最优（K-Best）方法”。它要求设置三个参数：

K：我们要求在某个接近最快值范围内的测量值数量。

ϵ：这些测量必须有多大程度的接近。也就是，如果测量值按照升序标号为 v_1，v_2，…，v_i，…，那么我们要求$(1+\varepsilon)v_1 \geq v_K$。

M：在我们中止之前，测量值的最大数量。

我们的实现进行了一系列尝试，并且按照排序的方式维护着一个 K 个最快时间的数组。对于每个新的测量值，它会检查这个值是否比当前数组位置 K 中的值更快。如果是，它会替换数组元素 K，然后执行一系列相邻数组位置之间的交换，将这个值移到数组中适当的位置。继续这个过程，直到误差标准满足，此时我们称测量值已经“收敛了”，或者我们超过了界限 M，此时我们称测量值不能收敛。

试验评价

我们进行了一系列试验来测量 K 次最优测量方法的准确性。下面是一些我们想要解决的问题：

1. 这个方法产生的是准确的测量值吗？
2. 什么时候测量值会收敛，收敛得有多快呢？
3. 这个方法能够确定它自己的测量值的准确性吗？

设计这样的试验的一个挑战是要知道我们正在试图测量的程序的实际运行时间。只有这样，我们才能确定我们测量的准确性。我们知道，只要我们正在测量的计算不被中断，我们的周期计时器就能够给出准确的结果。对于比计数器间隔[6]短很多的计算，运行在负载很轻的机器上时，被中断的可能性很小。我们利用这些属性来获得对真实运行时间的可靠估计值。

根据我们的测量目标，我们使用了一个过程，它反复地往一个 2048 个整数的数组中写值，然后再读出来，类似于 clear_cache 的代码。通过设置重复的次数 r，我们可以创建需要一定时间的计算。首先，我们设这个过程期望的运行时间为 r 的一个函数，用 T(r)来表示，r 从 1 变到 10，对运行时间计时（得到的时间为 0.09～0.9ms），执行最小二乘方拟合，找到形如 $T(r) = mr + b$ 的公式。通过使用小的 r 值，对每个 r 的值执行 100 次测量，并且在一个负载很轻的系统上运行测量，我们能够获得一个非常准确的 T(r)的描述。我们的最小二乘方分析表明公式 $T(r) = 49273.4r + 166$（单位为时钟周期）拟合这些数据，最大误差小于 0.04%。这使得我们有信心能够准确预测这个过程的实际计算时间，这个时间是 r 的一个函数。

然后，我们用 K 次最优方法来测量性能，参数 K = 3、$\epsilon = 0.001$，而 M = 30。我们对大量 r 的值进行这个测量，获得的预期运行时间的范围是 0.27～50ms。对于得到的每个测量值 M(r)，我们用 $E_m(r) = (M(r) - T(r))/T(r)$来计算测量误差 $E_m(r)$。图 9.14 展示了在一个 Intel Pentium III 上运行 Linux

6 实际上就是操作系统分配适合一个进程的时间段。——译者

的系统中，对 K 次最优方法的一个试验验证。在这张图中，我们给出了作为 T(r)的一个函数的测量误差 $E_m(r)$，这里我们给出的 T(r)的单位为 ms。注意，我们是以对数尺度来显示 $E_m(r)$的：每条水平线代表测量误差的一个数量级。为了使准确率在 1%之内，我们必须让误差在 0.01 以下。我们不试图显示任何小于 0.001（也就是 0.1%）的误差，因为我们的测试环境不提供这么高的精度。

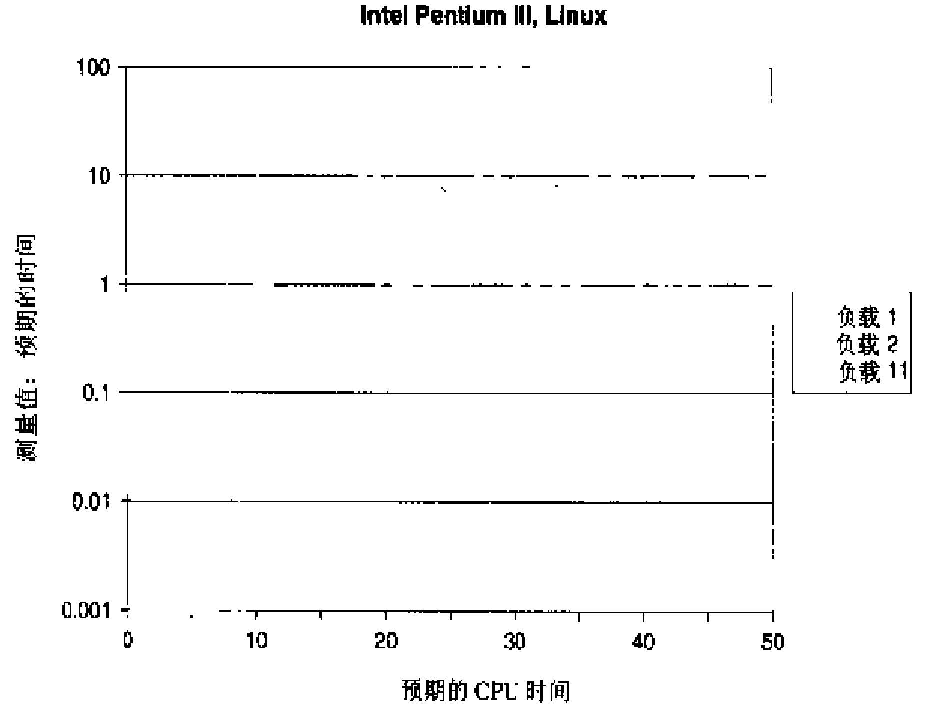


图 9.14　k 次最优测量方法在 Linux 系统上的验证

当执行时间最高为 8 ms 时，我们一直都能获得非常准确的测量值（误差大约在 0.1%）。而除此以外，在负载很轻的机器上，我们遇到的系统过高估计大约是 4%～6%，而在负载比较重的机器上，结果就更差。

这三组数据表明的是在三种不同负载情况下的误差。我们看到，在所有三种情况中，运行时间小于大约 7.5 ms 的测量都非常准确。因此，我们的方法可以用来在负载很重的机器上测量相对比较短的执行时间。“负载 1”那组数据表明的是只有一个活动进程的情况。对于大于 10ms 的执行时间，测量值 T_m 全都会过高估计计算时间 T_c 大约 4%～6%。过高估计是因为花了时间来处理计时器中断。这些数据与图 9.3 所示的 trace 一致，这个 trace 表明即使是在一台负载很轻的机器上，一个应用程序也只能执行 95%～96%的时间。“负载 2”和“负载 11”那两组数据展示的是还有其他进程在执行时的性能。在这两种情况中，对于超过大约 7 ms 的执行时间测量值不准确得离谱。注意，误差 1.0 就意味着 T_m 是 T_c 的两倍，而误差 10.0 就意味着 T_m 是 T_c 的 11 倍。很明显，操作系统调度每个活动进程一个计时器间隔 [7]。当有 n 个活动进程时，每个进程只获得 1/n 的处理器时间。

根据这些结果，我们可以得出结论说，K 次最优方法提供了对非常短时间计算的准确结果。对于测量超过大约 7 ms 的执行时间，这种方法真的不够好，特别是还有其他活动进程时。

不幸的是，我们发现我们的测量程序不能可靠地确定它是否获得了准确的测量值。我们的测量过程计算它的误差为 $E_p(r) = (v_k - v_1)/v_1$，这里 v_i 是第 i 个最小的测量值。也就是，它计算的是这个

7　原文是 time interval，而我们认为是 timer interval。

过程到达我们收敛标准的程度如何。我们发现这些估计值过于乐观了。即使是对于负载11的情况，测量值的偏移达到了10倍，程序却一直估计它的误差小于0.001。

设置K的值

在我们前面的试验中，我们任意选择参数K的值为3，即为了结束整个测量过程，在我们的测量结果中至少有3次的测量值相比于最快测量值间的误差在一个指定因子内。为了更仔细地衡量这个因素的影响，我们使K的值从1变化到5，并进行了一组测量，如图9.15所示。我们进行这些测量的执行时间范围到了9 ms，因为这是我们的方法能够获得有用结果的时间上限。

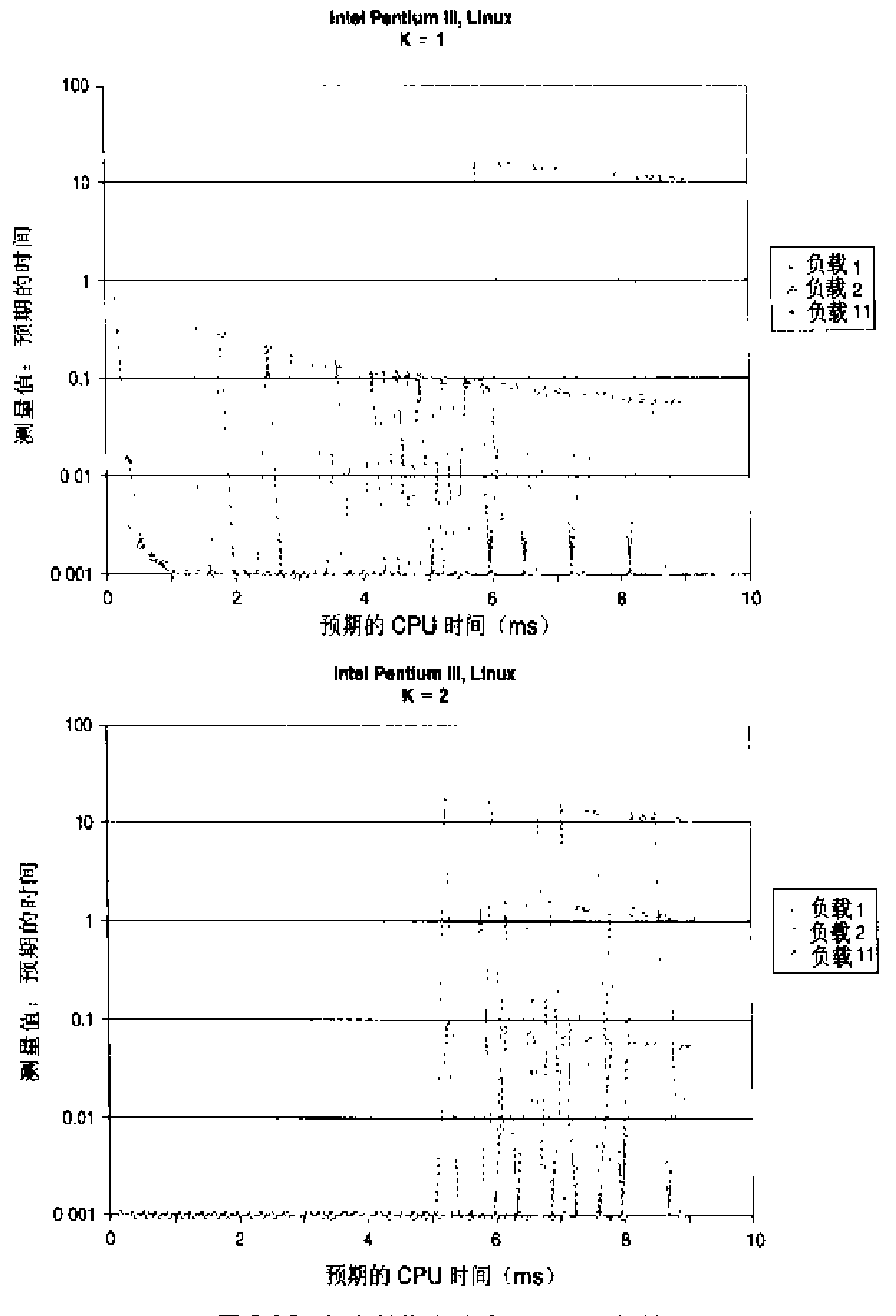


图9.15 k次最优方法中不同K值的效果

要想有合理的准确性，K必须至少为2。当程序时间接近于计时器间隔时，在负载很重的系统上，大于2的值会有帮助一些。

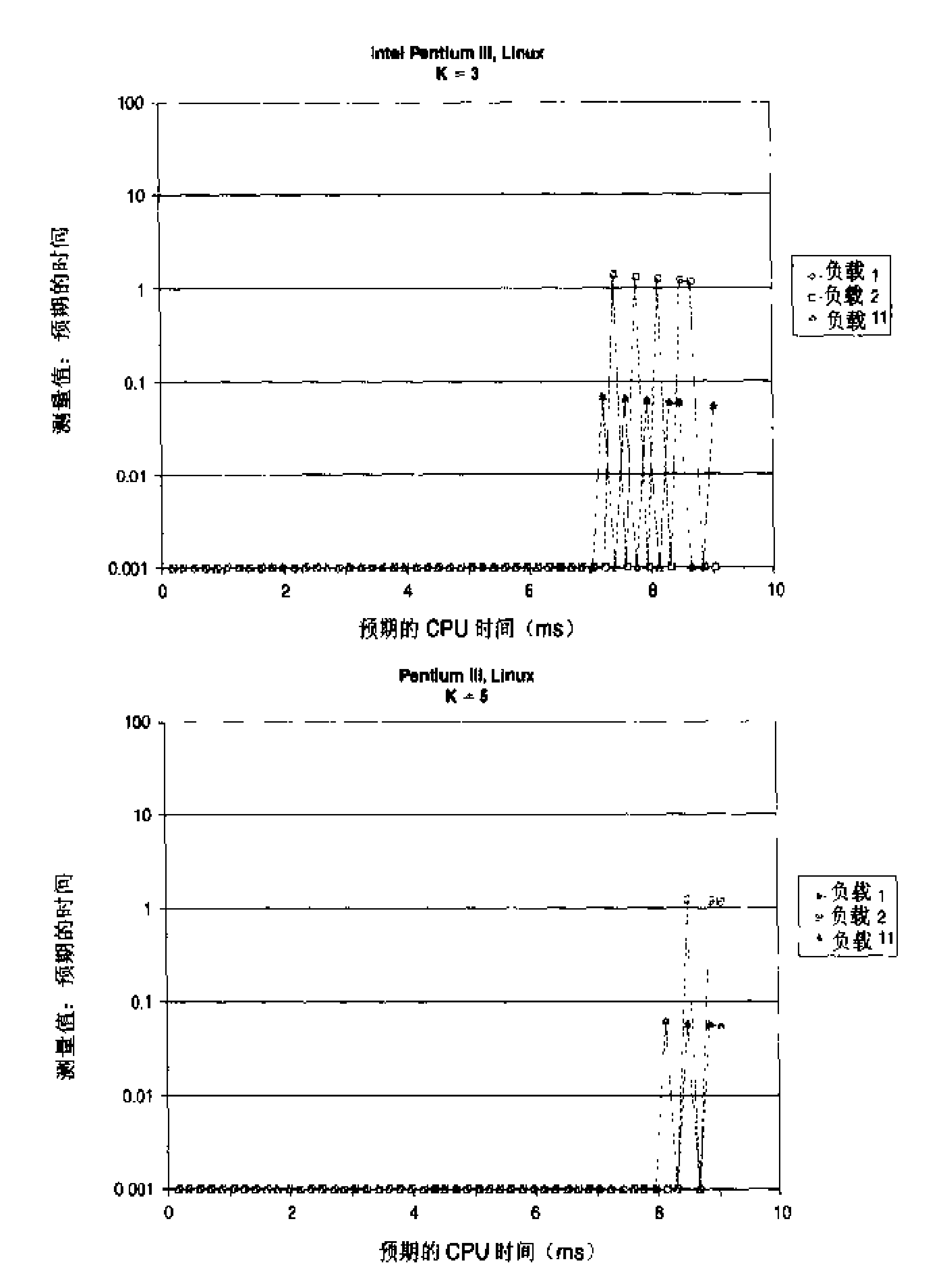


图 9.15 k 次最优方法中不同 K 值的效果（续）

当 K = 1 时，在进行一次测量后，过程就返回。这样得到的结果非常不规律，特别是当机器负载很重的时候。如果刚好发生了计时器中断，结果就更不准确了。即使没有发生这样的灾难，测量值也容易受很多因素的影响，变得不准确。将 K 设置为 2 就极大地改善了准确性。对于小于 5ms 的执行时间，我们得到的准确性都大于 0.1%。K 设置得越大，结果的一致性和准确性就越好，直到大约 8ms 的上限。这些试验表明我们最初猜想的 K = 3 是个合理的选择。

补偿对计时器中断的处理

计时器中断的发生是可预测的，在我们的执行时间超过大约 7 ms 的测量中，计时器中断会导致

很大的系统误差。通过从一个程序测量出的运行时间中减去一个花在处理计时器中断上的时间的估计值，除去这个偏差是很好的。这需要确定两个因素：

1. 我们必须确定处理一个计时器中断需要多少时间。为了保持我们从不低估过程的执行时间这一属性，我们应该确定处理一个计时器中断所需的最小时钟周期数。这样的话，我们永远不会过度补偿。

2. 我们必须确定在我们测量的时间段内发生了多少次计时器中断。

使用类似于产生图 9.3 和图 9.5 中所示 trace 的方法，我们可以发现不活动的时间段，并确定它们的持续时间。这些不活动时间段，有些是由计时器中断造成的，有些是由其他系统事件造成的。我们可以确定使用 times 过程是否会发生计时器中断，因为每次发生计时器中断时，它的返回值会增加一个滴答。我们对 100 个不活动周期进行这样一个评估，发现最小的计时器中断处理时间段需要 251 466 个周期。为了确定我们正在测量的程序执行期间发生的计时器中断次数，我们简单地调用 times 函数两次——一次在程序之前，一次在程序之后，然后计算它们的差。

图 9.16 展示了这种改进过的测量方法所获得的结果。如图中所示，现在在负载很轻的机器上，即使是对执行多个时间间隔的程序，我们也可以得到非常准确（在 1.0%之内）的测量值了。通过去掉计时器中断的系统误差，现在我们有了一个非常可靠的测量方法。另一方面，我们可以看到这种补偿对运行在负载很重的机器上的程序没有帮助。

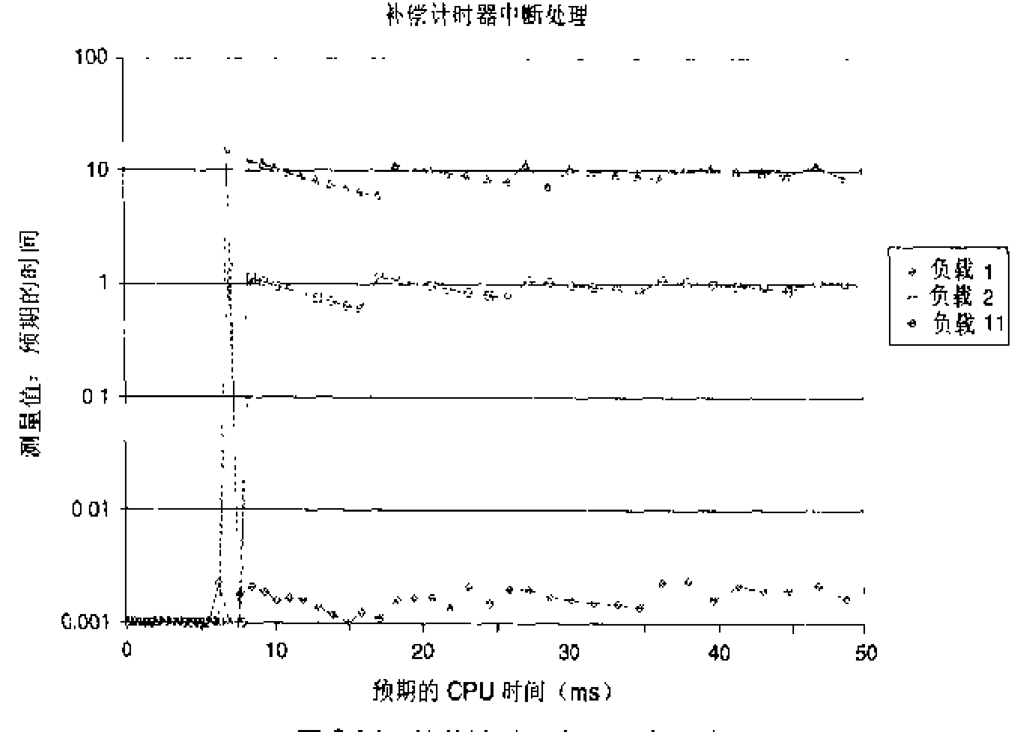


图 9.16 补偿计时器中断开销的测量

这种方法极大地提高了在负载很轻的机器上持续时间较长的测量的准确性。

在其他机器上的评估

因为我们的方法极大地依赖于操作系统的调度策略，所以我们还在其他三种系统配置上进行了试验：

1. 运行 Linux 内核较老版本（2.0.36 和 2.2.16）的 Intel Pentium III。

2. 运行 Windows-NT 的 Intel Pentium III。虽然这个系统使用的是 IA32 处理器，但是这个操作系统与 Linux 完全不同。

3. 运行 Tru64 Unix 的 Compaq Alpha。它使用的是一个非常不同的处理器，但是操作系统类似于 Linux。

如图 9.17 所示，在较老版本的 Linux 下的性能特性非常不同。在负载很轻的机器上，对于几乎任意持续时间的程序，测量值的准确性都在 0.2%以内。我们发现，使用这个版本的 Linux，处理器处理一个计时器中断只花费了大约 3 500 个周期。即使是在负载很重的机器上，它允许进程一次最多运行大约 180ms。这个试验表明操作系统的内部细节会极大地影响系统性能和我们获得准确测量值的能力。

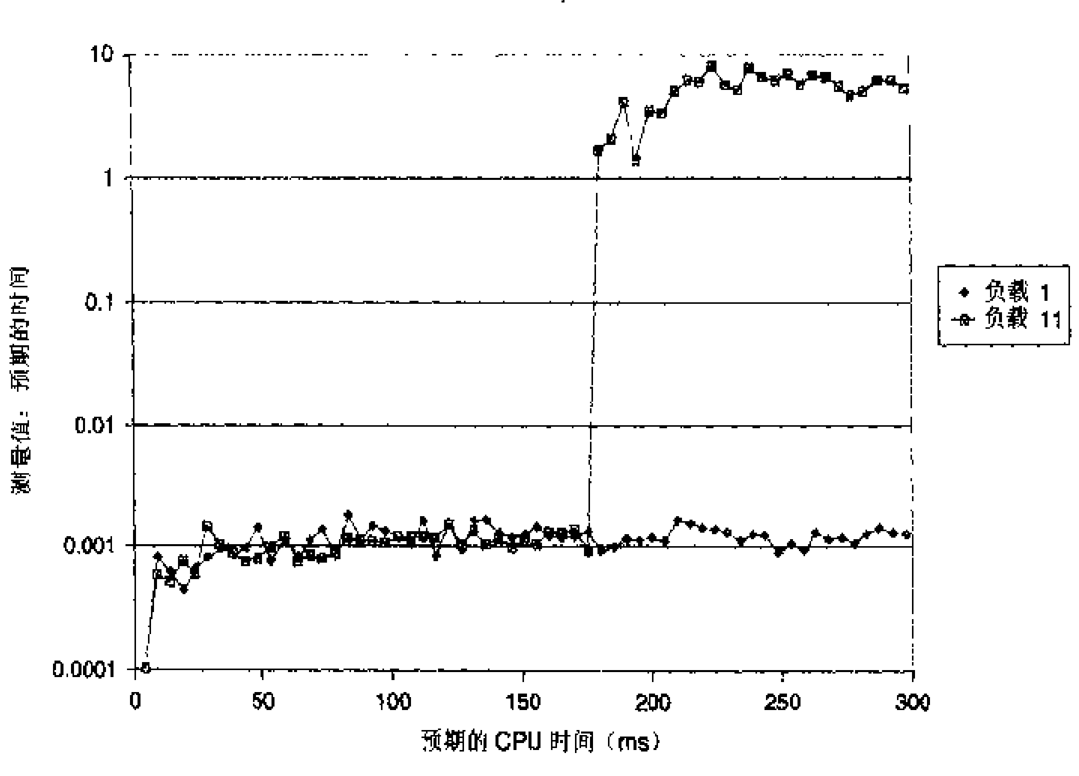


图 9.17 K 次最优测量方法在使用较老内核版本的 IA32/Linux 系统上的试验验证

在这个系统上，即使是对有较长执行时间的程序，特别是在负载很轻的机器上，我们也能获得更准确的测量值。

图 9.18 展示了在 Windows-NT 系统上的结果。总地来说，这些结果类似于那些较老 Linux 系统的结果。对于短时间的计算，或者在负载很轻的机器上，我们可以获得准确的测量值。在这种情况中，我们的准确度是大约 0.01（也就是 1.0%），而不是 0.001。不过，对于大多数应用来说，这就足够好了。另外，在负载很重的机器上，我们可靠的和不可靠的测量值之间的门限值大约是 48ms。一个有趣的特性是，有时候在负载很重的机器上，即使是对最长 245 ms 的计算，我们也能够获得准确的测量值。显然，NT 的调度器有时候会允许进程保持活动较长的一段时间，但是我们不能依靠这个属性。

Compaq Alpha 的结果如图 9.19 所示。我们再次发现，在负载很轻的机器上，几乎任意持续时间的程序测量出来的误差都小于 1.0%。在负载很重的机器上，只有持续时间小于大约 10ms 的程序才能被准确测量。

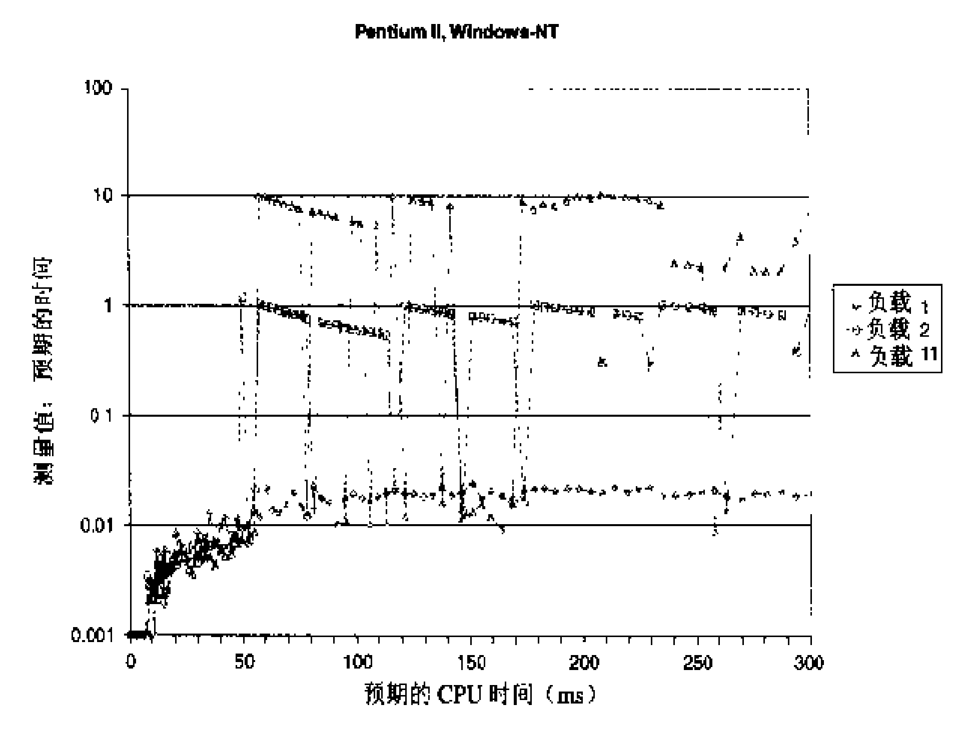


图 9.18 K 次最优测量方法在 Windows-NT 系统上的试验验证

在负载很轻的机器上，我们总是能获得准确的测量值（误差大约为 1.0%）。在负载很重的机器上，对于时长大于大约 48ms 的测量，准确性变得非常差。

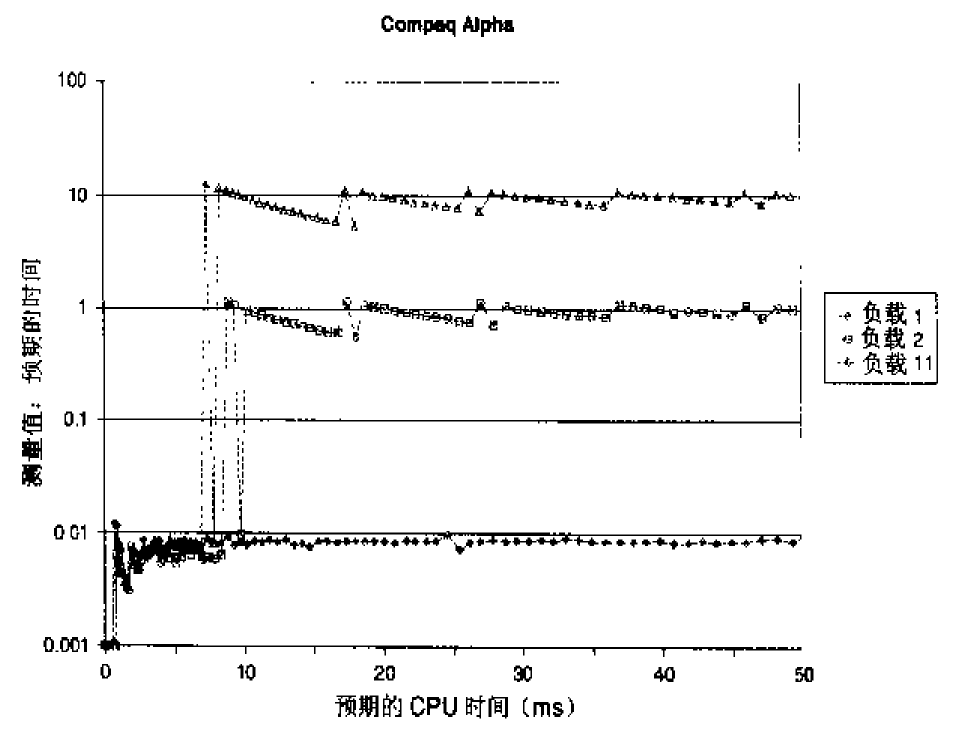


图 9.19 K 次最优测量方法在 Compaq Alpha 系统上的试验验证

对于负载很轻的系统，我们总是能获得准确（误差 < 1.0%）的测量值。对于负载很重的系统，大于 10ms 的持续时间就不能被准确地测量了。

练习题 9.7

假设我们希望测量一个需要 t ms 的过程。机器的负载很重，因此不允许我们的测量进程一次运

行超过 50ms。

A. 每次试验都包括测量这个过程的一次执行。假设一次试验从 50 ms 时间段中某个任意时间点开始，允许这次试验运行完成而不会被交换出来的概率是多少？将你的答案表示成 t 的一个函数，考虑所有可能的 t 的值。

B. 为了使其中的三次测量是这个过程的可靠测量（也就是，那些在一个时间段之内完成的运行），预期所需的测量次数是多少？将你的答案表示成 t 的一个函数。你预测 t = 20 和 t = 40 时这些值应该是多少？

观察

这些试验证明 K 次最优测量方法在多种机器上都工作得相当好。对于负载很轻的处理器，在大多数机器上，即使是对长持续时间的计算，它总是能得到准确的结果。只有较新版本的 Linux 会导致非常高的计时器中断开销，严重影响测量的准确性。对于这个系统，补偿这种开销会极大地提高测量的准确性。

在负载很重的机器上，当执行时间变得比较长时，获得准确的测量值变得很困难。大多数系统都有某个最大执行时间，当最大执行时间超出了测量界限，那么准确度会变得非常糟糕。这个门限值高度依赖于系统，但是通常是在 10～200ms 之间。

9.5 基于 gettimeofday 函数的测量

我们对 IA32 周期计数器的使用提供了高精度计时测量，但是它有个缺陷，那就是只能工作在 IA32 系统上。最好是有一个可移植性更好的解决方法。我们看到库函数 times 和 clock 是用间隔计数器来实现的，因此不是十分准确。

另一个可能性是使用库函数 gettimeofday。这个函数查询系统时钟（system clock）以确定当前的日期和时间。

```
#include "time.h"

struct timeval {
long tv_sec;        /* Seconds */
long tv_usec;       /* Microseconds */
}

int gettimeofday(struct timeval *tv, NULL);
```

返回：若成功则为 0，若失败则为-1.

这个函数把时间写入到一个调用者传递过来的结构中，这个结构包括一个单位为 s 的字段，还有一个单位为μs 的字段。第一个字段存放的是自从 1970 年 1 月 1 日以来经过的总秒数（对于所有的 Unix 系统来说，这都是一个标准的参考点）。注意，在 Linux 系统上，gettimeofday 的第二个参数，应该简单地置为 NULL，因为它指向一个未被实现的执行时区校正的特性。

练习题 9.8

在一个 32 位机器上，到什么日期 gettimeofday 写入到 tv_sec 字段的值会是负数？

如图 9.20 所示，我们可以用 gettimeofday 来创建两个计时器函数 start_timer 和 get_timer，它们

类似于我们的周期计时函数，除了它们的测量时间是以秒为单位，而不是以时钟周期为单位的。

code/perf/tod.c

```
1   #include <sys/time.h>
2   #include <unistd.h>
3
4   static struct timeval tstart;
5
6   /* Record current time */
7   void start_timer()
8   {
9       gettimeofday(&tstart, NULL);
10  }
11
12  /* Get number of seconds since last call to start_timer */
13  double get_timer()
14  {
15      struct timeval tfinish;
16      long sec, usec;
17
18      gettimeofday(&tfinish, NULL);
19      sec = tfinish.tv_sec - tstart.tv_sec;
20      usec = tfinish.tv_usec - tstart.tv_usec;
21      return sec + 1e-6*usec;
22  }
```

code/perf/tod.c

图 9.20 使用 Unix gettimeofday 的计时过程

这段代码可移植性非常好，但是它的准确性依赖于时钟是如何实现的。

这种计时机制依赖于 gettimeofday 是如何实现的，而 gettimeofday 的实现是随系统的不同而不同的。虽然函数产生一个以μs 为单位的测量值看上去非常好，但是事实证明测量值并不总是那么准确。图 9.21 展示了在几个不同的系统上测量这个函数的结果。我们定义函数的**分辨度**(resolution)为计时器可以分辨的最小时间值。我们通过反复调用 gettimeofday 直到写到第一个参数的值改变了，来计算这个值。那么，分辨度就是它改变了的μs 数。正如这张表所示，有些实现实际上可以分辨μs 级的时间，而另一些就没那么精确了。有这样一些差别，是因为有些系统用周期计数器来实现这个函数，而其他系统是用间隔计数的。在前者那种情况中，分辨度可以非常高——潜在地高于数据表示提供的 1ms 的分辨度。在后面那种情况中，分辨度会很糟糕——几乎和函数 times 和 clock 提供的相当。

图 9.21 还展示了在各种系统上调用 get_timer 所需的执行时间（latency）。这个属性表明了调用

这个函数所需要的最小时间。我们通过反复调用这个函数直到经过了 1s，再用 1 除以调用的次数，来计算这个值。正如看到的那样，在大多数系统上，这个函数调用需要大约 1ms，而在其他系统上需要几个 ms。相比之下，我们的过程 get_counter 每次调用只需要大约 0.2ms。一般而言，系统调用比普通的函数调用需要更多的开销。这个执行时间还限制了我们测量的精确度。即使数据结构允许以更高分辨度的单位来表达时间，但是当每次测量引起这么长时间的延迟时，我们还是不清楚能够多么准确地测量时间。

系　统	分辨度（μs）	执行时间（μs）
Pentium II, Windows-NT	10 000	5.4
Compaq Alpha	977	0.9
Pentium III Linux	1	0.9
Sun UltraSparc	2	1.1

图 9.21　gettimeofday 实现的特性

有些实现使用的是间隔计数，而其他的使用的是周期计时器。这极大地影响了测量的精确性。

图 9.22 展示了我们从一个使用 gettimeofday 而不是我们自己的函数来访问周期计数器的 K 次

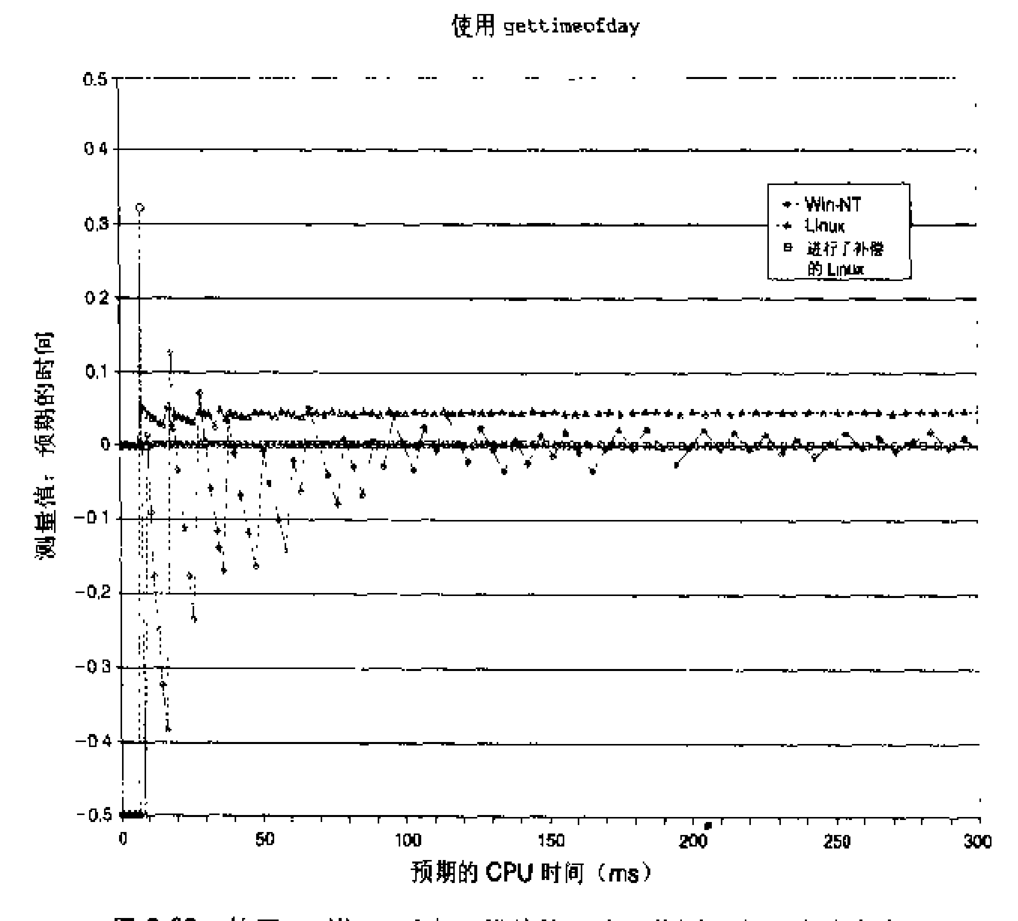


图 9.22　使用 gettimeofday 函数的 K 次最优测量方法的试验验证

Linux 是用周期计数器来实现这个函数的，所以精确度与我们的计时例程一样。Windows-NT 用间隔计数来实现这个函数的，因此精确度很低，特别是对于短的持续时间来说。

最优测量方法的实现得到的性能。我们展示了在两个不同机器上的结果，以说明时间分辨度对精确性的影响。在 Windows-NT 系统上的测量值表明的特性类似于我们在 Linux 系统上使用 times 时发现的特性（图 9.8）。因为 gettimeofday 是用进程计时器来实现的，所以误差可能是负的，也可能是正的，对于短持续时间的测量，尤其不规律。对于较长的持续时间，准确性有所改进，直到对于超过 200ms 的持续时间误差小于 2.0%时。在 Linux 系统上测量值给出的结果类似于直接使用周期计数器时得到的结果。通过比较图 9.14 中负载 1 结果的测量值（没有进行补偿）和图 9.16 中结果的测量值（进行了补偿），可以看出这一点。使用了补偿，即使是对长达 300ms 的测量，我们也可以获得好于 0.04%的准确度。因此，gettimeofday 与直接访问这台机器上的周期计数器完成得一样好。

9.6 综合：一个实验协议

我们可以以协议的形式总结一下我们的试验发现，来确定如何回答这个问题："程序 X 在机器 Y 上运行得有多快？"

- 如果 X 预期的运行时间很长（例如，大于 1.0s），那么间隔计数应该就工作得足够好了，而且对处理器负载不敏感。
- 如果 X 预期的运行时间在范围大约 0.01～1.0s 之间，那么在负载很轻的系统上，使用准确的、基于周期的计时来进行测量就很重要了。我们应该执行 gettimeofday 库函数的测量，来确定它在机器 Y 上的实现是基于周期的，还是基于间隔的。
  - 如果函数是基于周期的，那么用它作为 K 次最优计时函数的基础。
  - 如果函数是基于间隔的，那么我们必须找到一些使用机器的周期计数器的方法。这可能会需要汇编语言编码。
- 如果 X 预期的运行时间小于大约 0.01s，那么只要使用的是基于周期的计时，即使是在负载很重的机器上，也可以完成精确的测量。那么，我们着手用 gettimeofday 或直接访问机器的周期计数器，来实现一个 K 次最优计时函数。

9.7 展望未来

系统中引入了几个对性能测量有很大影响的特性：

- 与进程相关的周期计时。对于操作系统来说，管理周期计数器相对比较容易，所以它指明了某个进程经过的周期数。那么，当进程重新变为活动时，周期计数器被设置为当进程上次非活动（deactivated）时它的值，在进程不活动时有效地冻结了计数器。当然，计数器还是会受内核操作开销和高速缓存的影响的，但是至少其他进程的影响不会很严重。已经有一些系统支持这个特性了。根据我们的协议，这允许我们使用基于周期的计时来获得大于大约 0.01s 持续时间的准确测量值，即使是在负载很重的机器上。
- 频率变化的时钟。为了降低功耗，未来的系统会改变时钟频率，因为功耗直接与时钟频率相关。在那种情况中，我们不会有时钟周期与 ns 之间的一个简单的转换。甚至于很难知道应该用哪个单位来表达程序性能。对于代码优化器，通过计算周期，我们能获得

更多的了解，但是对于实现带实时性能限制的应用的人来说，实际运行时间会更重要一些。

9.8 现实生活：K 次最优测量方法

我们创建了一个库函数 fcyc，它使用 K 次最优方法来测量函数 f 所需要的时钟周期数：

```
#include "clock.h"
#include "fcyc.h"

typedef void (*test_funct)(int *);

double fcyc(test_funct f, int *params);
```

返回：运行参数 params 的函数 f 所使用的周期数。

参数 params 是一个指向整数的指针。一般而言，它可以指向一个整数数组，这个数组是被测量的函数的参数。例如，当测量大小写转换函数 lower1 和 lower2 时，我们传递一个指向一个 int 的指针作为参数，它是要转换的字符串的长度。在产生存储器山（第 6 章）时，我们可能要传递一个指向大小为 2 的数组的指针，其中包括大小和步长。

有很多控制测量的参数，例如 K、ε 和 M 的值，以及在每次测量之前是否要清除高速缓存。可以用同样在这个库中的函数来设置这些参数（详情请参见文件 fcyc.h）。

9.9 得到的经验教训

通过设计一种准确计时方法，以及在许多不同的系统上评价这种方法的性能的努力，我们学到了一些重要的经验：

- *每个系统都是不同的*。关于硬件、操作系统和库函数实现的细节对可以测量哪种程序以及精确度可以达到多少都有很大的影响。
- *试验可以是非常有启迪性的*。通过运行简单试验以产生活动 trace 的方法，我们获得了对操作系统调度程序的深入了解。这产生了补偿方法，它极大地提高了在负载很轻的 Linux 系统上的准确性。一个系统与另一个系统是不同的，即使是一个 OS 内核也与下一个版本的不同，能够分析和理解影响一个系统性能的各个方面是很重要的。
- *在负载很重的系统上获得准确的计时特别困难*。大多数系统研究者在专门的基准系统上进行他们所有的测量。他们常常关掉系统的许多 OS 和网络特性，以减少会引起不可预测活动的因素。不幸的是，普通的程序员没有这么奢侈。他们必须与其他用户共享系统。即使是在负载很重的系统上，我们的 K 次最优方法对于测量短于计时器间隔的持续时间来说，也是相当健壮的。
- *试验建立必须控制一些造成性能变化的因素*。高速缓存能够极大地影响一个程序的执行时

间。传统的技术是在计时开始之前，清空高速缓存中的所有有用的数据，或是在开始时，把通常会在高速缓存中的所有数据都加载进来。

9.10 小结

本章开始时提出了一个看似简单的问题："程序X在机器Y上运行得有多快？"不幸的是，计算机系统用来同时运行多个进程的机制使得很难获得程序性能可靠的测量值。系统活动倾向于在两个不同的时间尺度上进行。在微观级别上，每条指令执行的时间是以ns来衡量的。在宏观级别上，输入/输出交互发生的延迟是以ms来衡量的。计算机系统通过不断地从一个任务切换到另一个任务来利用这种差异，一次运行若干ms。

计算机系统有两种完全不同的记录时间流逝的方法。从宏观角度来看，计时器中断（timer interrupt）发生的频率似乎很快，但是从微观的角度来看却很慢。通过间隔计数（interval counting），系统能够获得对程序执行时间的非常粗略的测量值。这种方法只对长持续时间（至少1s）的测量有用。周期计数器（cycle counter）非常快，可以得到在微观尺度上很好的测量值。对于测量绝对时间的周期计数器，上下文切换的影响能够导致很小（在负载很轻的系统上）到很大（在负载很重的系统上）的误差。因此，没有方法是完美的。理解在一个特殊的系统上能够获得的准确度是很重要的。

取决于前面存储器引用和条件分支的历史，高速缓存和分支预测的影响可以导致执行代码的某个片段所需的时间每次都不同。通过事先运行某些将高速缓存设置为可预测状态的代码，我们可以部分地控制引起这种变化的因素，但是在有上下文切换发生时，这些尝试就没有用了。因此，我们必须进行多次测量，分析结果，以确定真实的执行时间。幸运的是，所有引起变化的因素的效果都是增加执行时间，因此只需分析确定测出的时间的最小值是否是一个准确的测量值。

通过一系列的试验，我们能够设计并且验证K次最优计时方法，这里我们反复进行测量，直到最快的K个值都在某个互相接近的范围之内了。在一些系统上，我们能够使测量用库函数来确定时间。在另一些系统上，我们必须通过汇编代码来访问周期计数器。

参考文献说明

关于程序计时的文献出奇得少。Stevens 的 Unix 编程著作[81]记录了程序计时的所有各种库函数。Wadleigh 和 Crawford 的关于软件优化的著作[85]描述了代码剖析和标准计时函数。

家庭作业

9.9 ◆◆

根据图9.3所示 trace 回答下列问题。我们的程序估计时钟频率为549.9MHz。然后，通过周期计数值来计算 trace 中的毫秒计时值。也就是说，对于一个以周期来表示的时间c，程序计算毫秒计时值为 c/549900。不幸的是，程序估计时钟频率的方法不完善，因此有些毫秒计时值不太准确。

A. 这个机器的计时器间隔为10ms。这些时间段中的哪些是由计时器中断发起的？

B. 根据这个 trace，操作系统服务一个计时器中断所需的最小时钟周期数是多少？

C. 根据这些 trace 数据，并且假设计时器间隔正好是 10.0ms，你推断真实的时钟频率是多少？

9.10 ◆◆

编写一个程序，它使用库函数 sleep 和 times 来确定每秒时钟滴答的近似次数。试着在多种系统上编译并运行这个程序。试着找出两个不同的系统，它们产生的结果至少相差两倍。

9.11 ◆

我们可以用周期计数器来生成活动的 trace，就像图 9.3 和 9.5 所示的那样。使用函数 start_counter 和 get_counter 来编写一个函数：

```
#include "clock.h"

int inactiveduration(int thresh);
                                                    返回：非活动的周期数。
```

这个函数不断地检查周期计数器，并察觉什么时候两个连续的读之间相差多于 thresh 个周期，这表明这个进程已经处于不活动状态了。返回这个不活动状态的持续时间（以时钟周期为单位）。

9.12 ◆

假设我们以参数 sleeptime 等于 2 调用函数 mhz(图 9.10)。系统的计数器间隔为 10ms。假设 sleep 是按照下面这样的方法来实现的。处理器维护一个计数器，每次发生计数器中断时，它都加 1。当系统执行 sleep(x)时，且当这个计数器达到 t + 100x 时，系统调度这个进程重新启动，这里 t 是计数器的当前值。

A. 设 w 表示由于调用 sleep，我们的进程处于不活动状态的时间。忽略函数调用、计时器中断等各种开销，w 的取值范围是多少？

B. 假设一次调用 mhz 得到 1000.0。再次忽略各种开销，真实的时钟频率可能的范围是多少？

练习题答案

练习题 9.1 答案

一开始，中断 CPU，并执行 100 000 个周期只为了处理一次击键，看上去很荒唐。不过，当你仔细研究一下这些数据，就会清楚 CPU 上的整个负载是相当轻的。

100 WPM 对应于每秒 10 次击键。100 个输入者每秒使用的周期总数会是 $10 \times 10^2 \times 10^5 = 10^8$，也就是处理器能够提供的总周期数的 10%。

练习题 9.2 答案

这个问题需要仔细地研究这个 trace，这样就会预期出模式的类型。

A. 它们每 9.98～9.99ms 发生一次：*358.93*，*368.91*，378.89，*388.88*，398.86，*408.85*，418.83，*428.81*。注意，没有用斜体表示的那些数字是由前面一个时间加上 9.98 得到的。

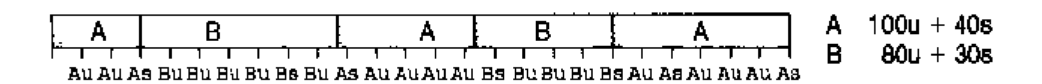
B. “A”中用斜体表示那些时间。它们引起一个新的不活动周期。

C. 除了花在执行其他进程上的时间以外，不活动时间还包括花在服务两个中断上的时间。

D. 我们的进程每 20.0ms 会活动大约 9.5ms，也就是总时间的 47.5%。

练习题 9.3 答案

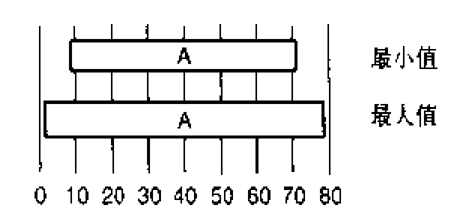
这道题就是简单地根据正在执行的进程标出执行顺序，并确定这个进程是在用户模式中还是在内核模式中。



练习题 9.4 答案

这是个很有趣的思考题。它帮助你推导出能够导致一个给定的间隔计数的可能的时间范围。

下图说明了两种情况：



对于最小的情况，片断刚好在时间 10 处的那个中断之前开始，刚好在时间 70 处的那个中断发生时结束，得到总时间刚好超过 60ms。对于最大的情况，片断刚好在时间 0 处的那个中断之后开始，并且一直持续到时间 80 处的那个中断之前结束，得到总时间刚好小于 80ms。

练习题 9.5 答案

这个习题要求思考的是记账（accouning）方法工作得如何。当进程是活动时，发生了 7 次计数器中断。在实际的 trace 中，进程在用户模式中运行了 63.7ms，而在内核模式中运行了 3.3ms。计数器过高地估计了真实的执行时间 70/(63.7 + 3.3) = 1.04X。

练习题 9.6 答案

这个习题要求推理出程序中各种导致延迟的因素，以及在什么情况中这些因素会起作用。

根据这些测量值，我们得到下列结论：

$$c + m + p + d = 399$$

$$c + d = 133 \pm 1$$

$$c + p = 317$$

根据这些结论，我们可以断定 $c = 100$、$d \approx 33$、$p = 217$，而 $m = 49$。

练习题 9.7 答案

这道题要求将概率论应用到一个简单的进程调度模型上。它说明当时间接近于进程时间极限时，获得准确的测量值变得非常困难。

A. 对于 $t \leqslant 50$，运行在一个时间段内的概率是 $1-t/50$。对于 $t > 50$，这个概率是 0。

B. 对于 $t \geqslant 50$，我们永远也不可能得到一次试验，它在一个进程时间段内执行完毕。对于 $t <$

50，成功的概率是 $p=(50-t)/50$，因此我们会期望 $3/p=150/(50-t)$ 次运行。对于 $t=20$，我们预期需要 5 次运行，而对于 $t=40$，我们预期需要 15 次。

练习题 9.8 答案

这是 Unix 版本的 Y2K 问题。有些人预测当时钟绕回来时会是一场全面的灾难。就像对待 Y2K 一样，我们相信这些恐惧是没有根据的。

这样的事情会在 1970 年 1 月 1 日后 2^{31} 秒后发生。那会是 2038 年 1 月 19 日凌晨 3:14。

CHAPTER 10

虚拟存储器

一个系统中的进程是与其他进程共享 CPU 和主存资源的。然而，共享主存会形成一些特殊的挑战。随着对 CPU 需求的增长，进程以某种合理的平滑方式慢了下来。但是如果太多的进程需要太多的存储器（memory），那么它们中的一些将简单地根本无法运行。当一个程序超出空间时，它就会成为那个运气不好的程序。

存储器还很容易被破坏。如果某个进程不小心写了另一个进程使用的存储器，那么进程可能以某种完全和程序逻辑无关的令人迷惑的方式失败。

为了更加有效地管理存储器并且少出错，现代系统提供了一种对主存的抽象概念，叫做虚拟存储器（VM）。虚拟存储器是硬件异常、硬件地址翻译、主存、磁盘文件和内核软件的完美交互，它为每个进程提供了一个大的、一致的、私有地址空间。通过一个很清晰的机制，虚拟存储器提供了三个重要的能力：它将主存看成是一个存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传送数据，通过这种方式，它高效地使用了主存；它为每个进程提供了一致的地址空间，从而简化了存储器管理；它保护了每个进程的地址空间不被其他进程破坏。

虚拟存储器是计算机系统最重要的概念之一。它成功的一个主要原因就是因为它是沉默地、自动地工作的，不需要应用程序员的任何干涉。既然虚拟存储器在幕后工作得如此之好，为什么程序员还需要理解它呢？有以下几个原因：

- *虚拟存储器是中心的。*虚拟存储器遍及计算机系统的所有层面，在硬件异常、汇编器、链接器、加载器、共享对象、文件和进程的设计中扮演着重要角色。理解虚拟存储器将帮助你更好地理解系统通常是如何工作的。
- *虚拟存储器是强大的。*虚拟存储器给予应用程序强大的能力，可以创建和破坏存储器块、将存储器块映射到磁盘文件的某个部分，以及与其他进程共享存储器。比如，你知道你可以通过读写存储器位置读或者修改一个磁盘文件的内容吗？或者是你可以加载一个文件的内容到存储器中，而不需要进行任何显式地拷贝吗？理解虚拟存储器将帮助你利用它的强大功能在你的应用程序中添加动力。
- *虚拟存储器是危险的。*每次应用程序引用一个变量、间接引用一个指针，或者调用一个诸如 malloc 这样的动态分配包程序时，它就会和虚拟存储器发生交互。如果虚拟存储器使用不当，应用将遇到复杂险恶的与存储器有关的错误。例如，一个带有错误指针的程序可以立即崩溃于“段错误”或者“保护错误”，它可能在崩溃之前还默默地运行了几个小时，或者是最令人惊慌地，运行完成，却产生不正确的结果。理解虚拟存储器，以及诸如 malloc 之类的管理虚拟存储器的分配程序包，可以帮助你避免这些错误。

这一章从两个角度来讨论虚拟存储器。本章的前一部分描述虚拟存储器是如何工作的，后一部分描述的是应用程序如何使用和管理虚拟存储器。无可避免的事实是虚拟存储器很复杂，本章很多地方都反映了这一点。好消息就是如果你掌握这些细节，你就能够手工模拟一个小系统的虚拟存储器机制，而且虚拟存储器的概念将永远不再神秘。

第二部分是建立在这种理解之上的，向你展示了如何在程序中使用和管理虚拟存储器。你将学会如何通过显式的存储器映射和对像 malloc 程序包这样的动态存储分配程序的调用，来管理虚拟存储器。你还将了解到 C 程序中的一大群常见的与存储器有关的错误，并学会如何避免它们的出现。

10.1 物理和虚拟寻址

计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个惟一的物理地址（physical address，PA）。第一个字节的地址为 0，接下来的字节地址为 1，再下一个为 2，依此类推。给定这种简单的结构，CPU 访问存储器的最自然的方式就是使用物理地址。我们把这种方式称为物理寻址（physical addressing）。图 10.1 展示了一个物理寻址的示例，该示例的上下文是一条加载指令，读取从物理地址 4 处开始的字。

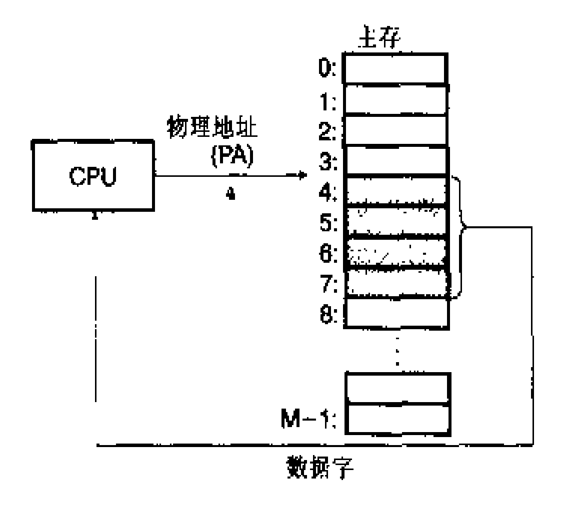


图 10.1 一个使用物理寻址的系统

当 CPU 执行这条加载指令时，它会生成一个有效的物理地址，通过存储器总线，把它传递给主存。主存取出从物理地址 4 处开始的 4 字节的字，并将它返回给 CPU，CPU 会将它存放在一个寄存器里。

早期的 PC 使用物理寻址，而且诸如数字信号处理器、嵌入式微控制器以及 Cray 超级计算机这样的系统仍然继续使用这种寻址方式。然而，为通用计算设计的现代处理器使用的是虚拟寻址（virtual addressing），参见图 10.2。

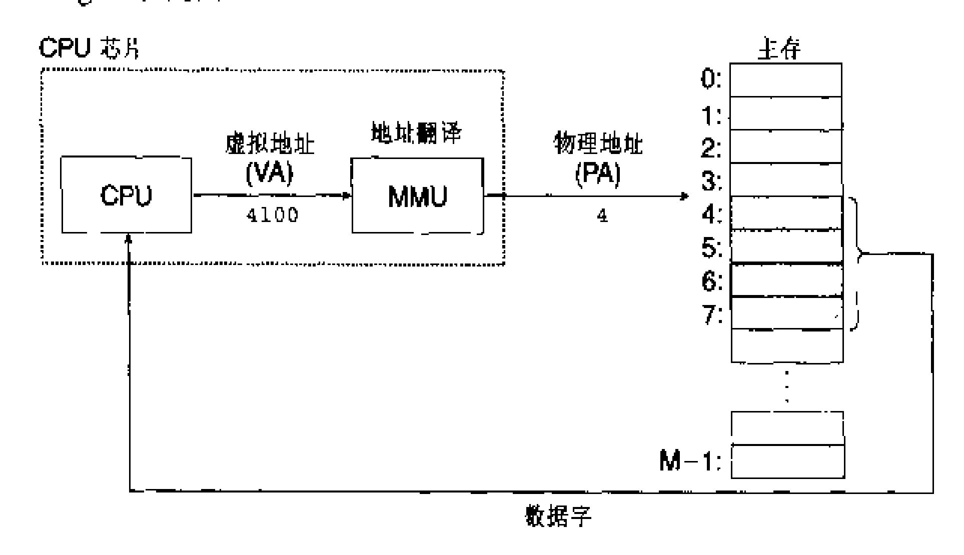


图 10.2 一个使用虚拟寻址的系统

根据虚拟寻址，CPU 通过生成一个*虚拟地址*（virtual address，VA）来访问主存，这个虚拟地址在被送到存储器之前先转换成适当的物理地址。将一个虚拟地址转换为物理地址的任务叫做*地址翻译*（address translation）。就像异常处理一样，地址翻译需要 CPU 硬件和操作系统之间的紧密合作。CPU 芯片上叫做 MMU（memory management unit，*存储器管理单元*）的专用硬件，利用存放在主存中的查询表来动态翻译虚拟地址，该表的内容由操作系统管理的。

10.2　地址空间

地址空间（address space）是一个非负整数地址的有序集合：

$$\{0, 1, 2, \cdots\}$$

如果地址空间中的整数是连续的，那么我们说它是一个*线性地址空间*（linear address space）。为了简化我们的讨论，我们总是假设使用的是线性地址空间。在一个带虚拟存储器的系统中，CPU 从一个有 $N=2^n$ 个地址的地址空间中生成虚拟地址，这个地址空间称为*虚拟地址空间*（virtual address space）：

$$\{0, 1, 2, \cdots, N-1\}。$$

一个地址空间的大小是由表示最大地址所需要的位数来描述的。例如，一个包含 $N=2^n$ 个地址的虚拟地址空间就叫做一个 n 位地址空间。现代系统典型地都支持 32 位或者 64 位虚拟地址空间。

一个系统还有一个*物理地址空间*（physical address space），它与系统中物理存储器的 M 个字节相对应：

$$\{0, 1, 2, \cdots, M-1\}。$$

M 不要求是 2 的幂，但是为了简化讨论，我们假设 $M=2^m$。

地址空间的概念是很重要的，因为它清楚地区分了数据对象（字节）和它们的属性（地址）。一旦我们认识到了这种区别，那么我们就可以概括总结，允许每个数据对象有多个独立的地址，其中每个地址都选自一个不同的地址空间。这就是虚拟存储器的基本思想。主存中的每字节都有一个选自虚拟地址空间的虚拟地址，和一个选自物理地址空间的物理地址。

练习题 10.1

完成下面的表格，填写缺失的条目，并且用适当的整数取代每个问号。利用下列单位：$K=2^{10}$（千），$M=2^{20}$（兆，百万），$G=2^{30}$（千兆，十亿），$T=2^{40}$（万亿），$P=2^{50}$（千千兆），$E=2^{60}$（千兆兆）。

#虚拟地址位数（n）	#虚拟地址数（N）	最大可能的虚拟地址
8		
	$2^{?}=64K$	
		$2^{32}-1=?G-1$
	$2^{?}=256T$	
64		

10.3 虚拟存储器作为缓存的工具

概念上而言，虚拟存储器（VM）被组织为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每字节都有一个惟一的虚拟地址，这个惟一的虚拟地址是作为到数组的索引的。磁盘上数组的内容被缓存在主存中。和存储器层次结构中其他缓存一样，磁盘（较低层）上的数据被分割成块，这些块作为磁盘和主存（较高层）之间的传输单元。VM 系统通过将虚拟存储器分割为称为*虚拟页*（virtual page，VP）的大小固定的块，来处理这个问题。每个虚拟页的大小为 $P=2^p$ 字节。类似地，物理存储器被分割为*物理页*（physical page，PP），大小也为 P 字节（物理页也被称为页帧，page frame）。

在任意时刻，虚拟页面的集合都分为三个不相交的子集：

- **未分配的**：VM 系统还未分配（或者创建）的页。未分配的块没有任何数据和它们相关联，因此也就不占用任何磁盘空间。
- **缓存的**：当前缓存在物理存储器中的已分配页。
- **未缓存的**：没有缓存在物理存储器中的已分配页。

图 10.3 的示例展示了一个有 8 个虚拟页的小虚拟存储器。虚拟页 0~3 还没有被分配，因此在磁盘上还不存在。虚拟页 1、4 和 6 被缓存在物理存储器中。页 2、5 和 7 已经被分配了，但是当前并未缓存在主存中。

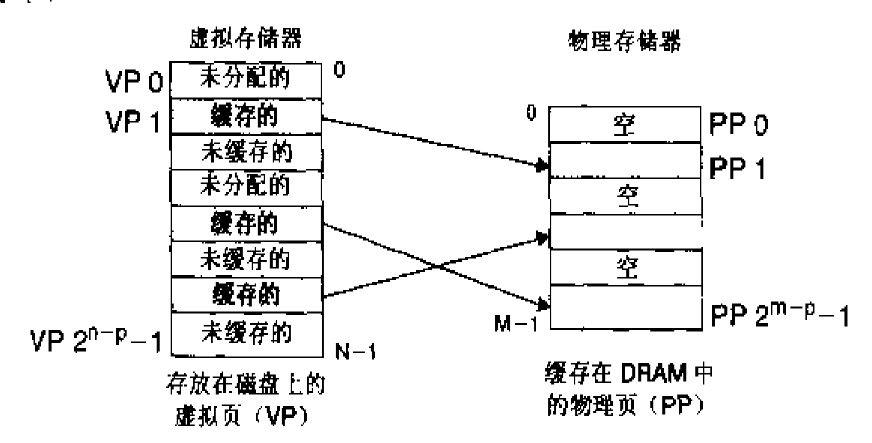


图 10.3 一个虚拟存储器系统是如何使用主存作为缓存的

10.3.1 DRAM 高速缓存的组织结构

为了帮助我们清晰理解存储层次结构中不同的缓存概念，我们将使用术语 SRAM 缓存来表示位于 CPU 和主存之间的 L1 和 L2 高速缓存，并且用术语 DRAM 缓存来表示虚拟存储器系统的缓存，它在主存中缓存虚拟页。

在存储层次结构中，DRAM 缓存的位置对它的组织结构有很大的影响。回想一下：DRAM 比 SRAM 要慢大约 10 倍，而磁盘要比 DRAM 慢大约 100 000 多倍。因此，DRAM 缓存中的不命中（miss）比起 SRAM 缓存中的不命中要昂贵得多，因为 DRAM 缓存不命中要由磁盘来服务，而 SRAM 缓存不命中通常是由基于 DRAM 的主存来服务的。而且，从磁盘的一个扇区读取第一字节的时间开销比起读这个扇区中后面的字节要慢大约 100 000 倍。归根到底，DRAM 缓存的组织结构完全是由巨大的不命中开销驱动的。

因为大的不命中处罚和访问第一字节的开销，虚拟页趋向于很大，典型地是4～8KB。由于大的不命中处罚，DRAM缓存是全相联的，也就是说，任何虚拟页都可以放置在任何的物理页中。不命中时的替换策略也很重要，因为替换错了虚拟页的处罚也非常之高。因此，比起硬件对SRAM缓存，操作系统对DRAM缓存使用了更复杂精密的替换算法（这些替换算法超出了我们的讨论范围）。最后，因为对磁盘的访问时间很长，DRAM缓存总是使用写回（write-back），而不是直写（write-through）。

10.3.2 页表

同任何缓存一样，虚拟存储器系统必须有某种方法来判定一个虚拟页是否存放在DRAM中的某个地方。如果是，系统还必须确定这个虚拟页存放在哪个物理页中。如果不命中，系统必须判断这个虚拟页存放在磁盘的哪个位置，在物理存储器中选择一个牺牲页，并将虚拟页从磁盘拷贝到DRAM中，替换这个牺牲页。

这些功能是由许多软硬件联合提供的，包括：操作系统软件、MMU（存储器管理单元）中的地址翻译硬件，和一个存放在物理存储器中叫做页表（page table）的数据结构。页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。操作系统负责维护页表的内容，以及在磁盘与DRAM之间来回传送页。

图10.4展示了一个页表的基本组织结构。页表就是一个PTE（page table entry，页表条目）的数组。虚拟地址空间中的每个页在页表中的一个固定偏移量处都有一个PTE。为了我们的目的，我们将假设每个PTE是由一个有效位（valid bit）和一个n位地址字段组成的。有效位表明了该虚拟页当前是否被缓存在DRAM中。如果设置了有效位，那么地址字段就表示DRAM中相应的物理页的起始位置，这个物理页中缓存了该虚拟页。如果没有设置有效位，那么有一个空地址表示这个虚拟页还未被分配。否则，这个地址就指向磁盘上虚拟页的起始位置。

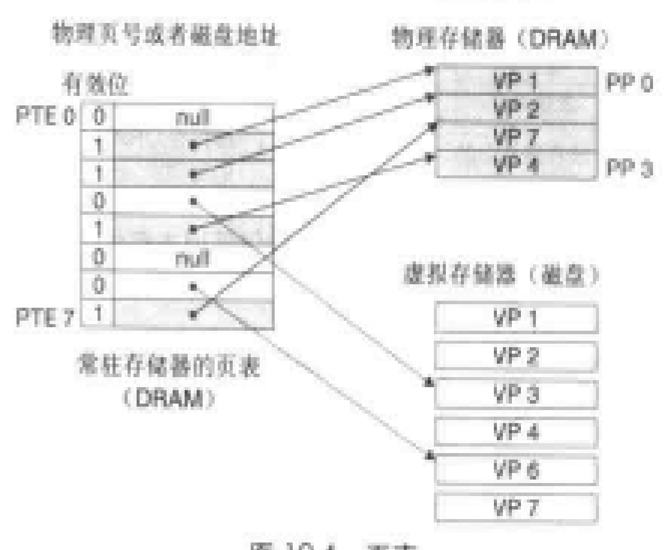


图10.4 页表

图10.4中的示例展示了一个有8个虚拟页和4个物理页的系统的页表。四个虚拟页（VP1、VP2、VP4和VP7）当前被缓存在DRAM中。两个页（VP0和VP5）还未被分配，而剩下的页（VP3和VP6）已经被分配了，但是当前还未被缓存。图10.4中有一个要点要注意，因为DRAM缓存是全相联的，任意物理页都可以包含任意虚拟页。

练习题 10.2

确定下列虚拟地址大小（n）和页大小（P）的组合所需要的 PTE 数量：

n	$P = 2^p$	# PTE
16	4K	
16	8K	
32	4K	
32	8K	

10.3.3 页命中

考虑一下当 CPU 读虚拟存储器的一个字时，它被 VP2 包含且被缓存在 DRAM 中，会发生什么（参见图 10.5）。使用我们将在 10.6 节中详细描述的一种技术，地址翻译硬件将虚拟地址作为一个索引，来定位 PTE2，并从存储器中读取它。既然设置了有效位，那么地址翻译硬件就知道 VP2 是缓存在存储器中的了，所以它使用 PTE 中的物理存储器地址（该地址指向 PP0 中缓存页的起始位置），构造出这个字的物理地址。

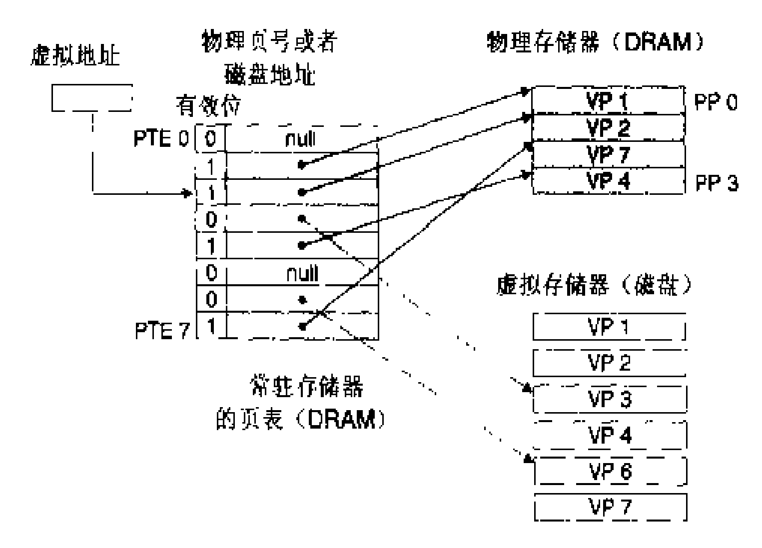


图 10.5 VM 页命中

对 VP2 中一个字的引用就命中了。

10.3.4 缺页

在虚拟存储器的习惯说法中，DRAM 缓存不命中称为缺页（page fault）。图 10.6 展示了在缺页之前我们的示例页表的状态。CPU 引用了 VP3 中的一个字，这个字并未缓存在 DRAM 中。地址翻译硬件从存储器中读取 PTE3，从有效位推断出 VP3 未被缓存，并且触发一个缺页异常。

缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在此例中就是存放在 PP3 中的 VP4。如果 VP4 已经被修改了，那么内核就会将它拷贝回磁盘。无论哪种情况，内核都会修改 VP4 的页表条目，反映出 VP4 不再缓存在主存中这一事实。

接下来，内核从磁盘拷贝 VP3 到存储器中的 PP3，更新 PTE3，随后返回。当异常处理程序返回时，它会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬件。但是现在，VP3 已经缓存在主存中了，那么页命中也能由地址翻译硬件正常处理了，就像我们在图 10.5

中看到的那样。图10.7展示了在缺页之后我们的示例页表的状态。

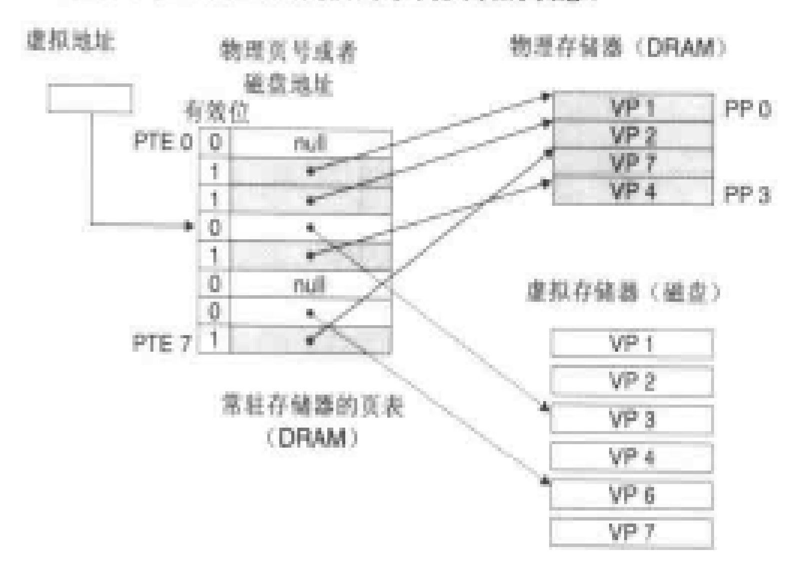


图10.6 VM缺页(之前)

对VP3中的字的引用不命中，从而触发了缺页。

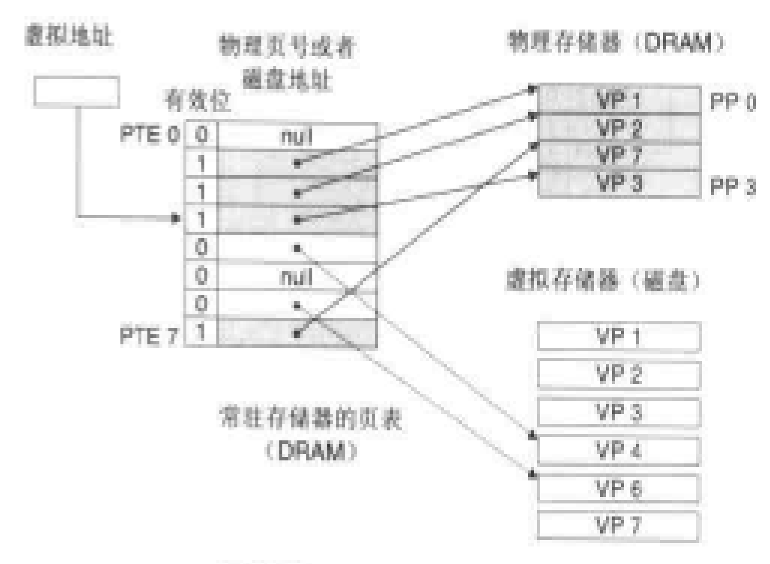


图10.7 VM缺页(之后)

缺页处理程序选择VP4作为牺牲页，并从磁盘上用VP3的拷贝取代它。在缺页处理程序重新启动导致缺页的指令之后，该指令将从存储器中正常地读取字，而不会再产生异常。

虚拟存储器是在20世纪60年代早期发明的，远在CPU-存储器之间差距的加大引发产生SRAM缓存之前。因此，虚拟存储器系统使用了和SRAM缓存不同的术语，即使它们的许多概念是相似的。在虚拟存储器的习惯说法中，块被称为页。在磁盘和存储器之间传送页的活动叫做交换(swapping)或者页面调度(paging)。页从磁盘换入(或者页面调入)DRAM，和从DRAM换出到(或者页面调出到)磁盘。一直等待，直到最后时刻，也就是当有不命中发生时，才换入页面的这种策略被称为按需页面调度(demand paging)。其他的方法也是可能的，例如尝试着预测不命中，在页面实际被引用之前就换入页面。然而，所有现代系统都使用的是按需页面调度的方式。

10.3.5 分配页面

图 10.8 展示了当操作系统分配一个新的虚拟存储器页时，对我们示例页表的影响，例如，调用 malloc 的结果。在这个示例中，通过在磁盘上创建空间，并更新 PTE5，使它指向磁盘上这个新创建的页面，从而分配 VP5。

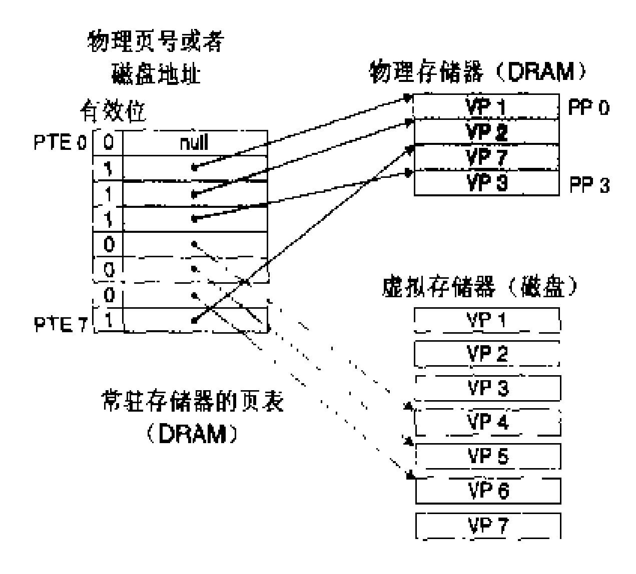


图 10.8 分配一个新的虚拟页面

内核在磁盘上分配 VP5，并且将 PTE5 指向这个新的位置。

10.3.6 局部性再次搭救

当我们中的许多人都了解了虚拟存储器的概念之后，我们的第一印象通常是它的效率想必是非常低。假设不命中处罚很大，我们会担心页面调度会破坏程序性能。实际上，虚拟存储器工作得相当好，这主要归功于我们的老朋友*局部性*（locality）。

尽管在整个运行过程中程序引用的不同页面的总数可能超出物理存储器总的大小，但是局部性原则保证了在任意时刻，这些页面将趋向于在一个较小的*活动页面*（active page）集合上工作，这个集合叫做*工作集*（working set）或者*常驻集合*（resident set）。在初始开销，也就是将工作集页面调度到存储器中，之后，接下来对这个工作集的引用将导致命中，而不会产生额外的磁盘流量。

只要我们的程序有好的时间局部性，虚拟存储器系统就能工作得相当好。但是，当然，不是所有的程序都能展现良好的时间局部性。如果工作集的大小超出了物理存储器的大小，那么程序将产生一种不幸的状态，叫做*颠簸*（thrashing），这时页面将不断地换进换出。虽然虚拟存储器通常是有效的，但是如果一个程序性能慢得像爬一样，那么聪明的程序员会考虑看是不是发生了颠簸。

旁注：统计缺页次数

你可以利用 Unix 的 getrusage 函数监测缺页的数量（以及许多其他的信息）。

10.4 虚拟存储器作为存储器管理的工具

在上一节中，我们看到虚拟存储器是如何提供一种机制，利用 DRAM 来缓存来自通常更大的虚

拟地址空间的页面。有趣地是，一些早期的系统，比如 DEC PDP-11/70，支持的是一个比物理存储器更小的虚拟地址空间。然而，虚拟地址仍然是一个有用的机制，因为它大大地简化了存储器管理，并提供了一种简单自然的保护存储器的方法。

到目前为止，我们都假设有一个单独的页表，将一个虚拟地址空间映射到物理地址空间。实际上，操作系统为每个进程提供了一个独立的页表，因而也就是一个独立的虚拟地址空间。图 10.9 展示了基本概念。在这个示例中，进程 i 的页表将 VP1 映射到 PP2，VP2 映射到 PP7。相似地，进程 j 的页表将 VP1 到 PP7，VP2 映射到 PP10。注意，多个虚拟页面可以映射到同一个共享物理页面上。

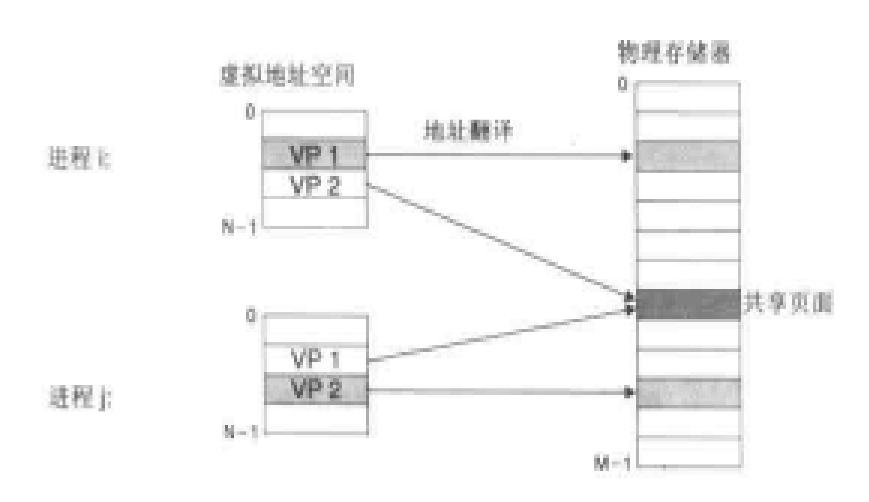


图 10.9 VM 如何为进程提供独立的地址空间

操作系统为系统中的每个进程都维护一个独立的页表。

按需页面调度和独立的虚拟地址空间的结合对系统中存储器的使用和管理造成了深远的影响。特别地，VM 简化了链接和加载，共享代码和数据，以及对应用分配存储器。

10.4.1 简化链接

独立的地址空间允许每个进程为它的存储器映像使用相同的基本格式，而不管代码和数据实际存放在物理存储器的何处。例如，每个 Linux 进程都使用图 10.10 所示的格式。

文本区总是从虚拟地址 0x08048000 处开始，栈总是从地址 0xbfffffff 向下伸展，共享库代码总是从地址 0x40000000 处开始，而操作系统代码和数据总是从地址 0xc0000000 开始。这样的一致性极大地简化了链接器的设计和实现，允许链接器生成全链接的可执行文件，这些可执行文件是独立于物理存储器中代码和数据的最终位置的。

10.4.2 简化共享

独立地址空间为操作系统提供了一个管理用户进程和操作系统自身之间共享的一致机制。一般而言，每个进程都有自己私有的代码、数据、堆以及栈区域，是不和其他进程共享的。在这种情况中，操作系统创建页表，将相应的虚拟页映射到不同的物理页面。

然而，在一些情况中，还是需要进程来共享代码和数据。例如，每个进程必须调用相同的操作系统内核代码，而每个 C 程序都会调用标准库中的程序，比如 printf。操作系统通过将不同进程中适当的虚拟页面映射到相同的物理页面，从而安排多个进程共享这部分代码的一个拷贝，而不是在

每个进程中都包括单独的内核和C标准库的拷贝。

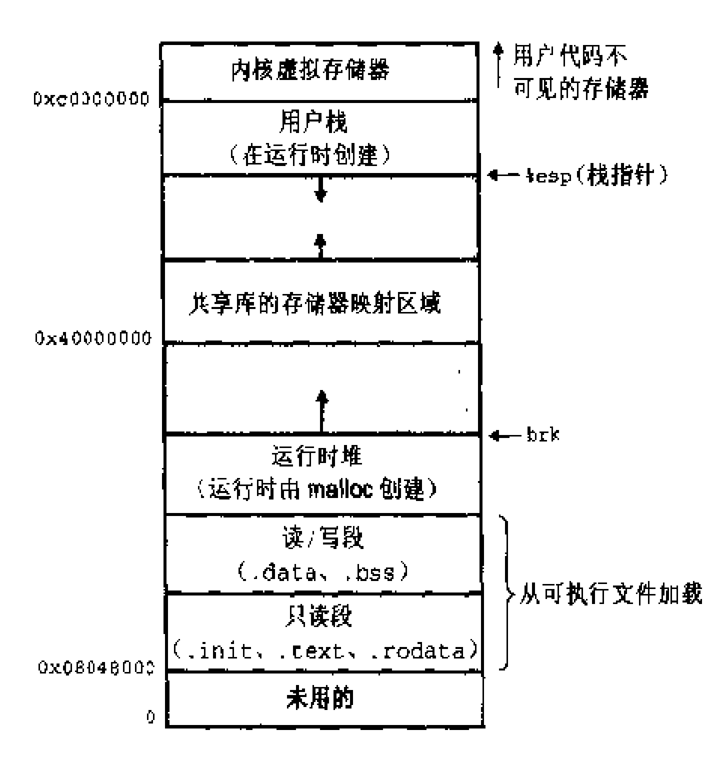


图 10.10 一个Linux进程的存储器映像

程序总是从虚拟地址 0x8048000处开始。用户栈总是从虚拟地址0xbfffffff处开始。共享对象总是加载在从虚拟地址0x40000000处开始的区域内。

10.4.3 简化存储器分配

虚拟存储器为向用户进程提供一个简单的分配额外存储器的机制。当一个运行在用户进程中的程序要求额外的堆空间时（例如，调用 malloc 的结果），操作系统分配一个适当数字（例如k）个连续的虚拟存储器页面，并且将它们映射到物理存储器中任意位置的k个任意的物理页面。由于页表工作的方式，操作系统没有必要分配k个连续的物理存储器页面。页面可以随机地分散在物理存储器中。

10.4.4 简化加载

虚拟存储器也使加载可执行文件和已共享目标文件到存储器中变得容易。回想一下，ELF可执行文件中的.text 和.data 节是相邻的。为了加载这些节到一个新创建的进程中，Linux 加载程序分配了一个从地址 0x08048000 处开始的连续的虚拟页面区域，将它们标识为无效的（也就是未缓存的），并将它们的页表条目指向目标文件中适当的位置。

有趣的一点是加载器从不真正地从磁盘中拷贝任何数据到存储器中。当每个页面第一次被引用时，虚拟存储器系统将自动并按需地把数据从磁盘上调入到存储器，页面引用或者是当CPU取一条指令时，或者是当一条正在执行的指令引用一个存储器位置时。

映射一个连续虚拟页面的集合到任意一个文件中的任意一个位置的概念叫做存储器映射（memory mapping）。Unix 提供了一个叫做 mmap 的系统调用，允许应用程序进行自己的存储器映射。我们将在 10.8 节中更详细地描述应用层存储器映射。

10.5 虚拟存储器作为存储器保护的工具

任何现代计算机系统必须为操作系统提供手段来控制对存储器系统的访问。不应该允许一个用户进程修改它的只读文本段，而且也不应该允许它读或修改任何内核中的代码和数据结构。不应该允许它读或者写其他进程的私有存储器，并且不允许它修改任何与其他进程共享的虚拟页面，除非所有的共享者都显式地允许它这么做（通过调用明确的进程间通信系统调用）。

就像我们所看到的，提供独立的地址空间使得分离不同进程的私有存储器变得容易。但是，地址翻译机制可以以一种自然的方式扩展到提供更好的访问控制。因为每次 CPU 生成一个地址时，地址翻译硬件都会读一个 PTE，所以通过在 PTE 上添加一些额外的许可位来控制对一个虚拟页面内容的访问，十分简单。图 10.11 展示了一般的概念。

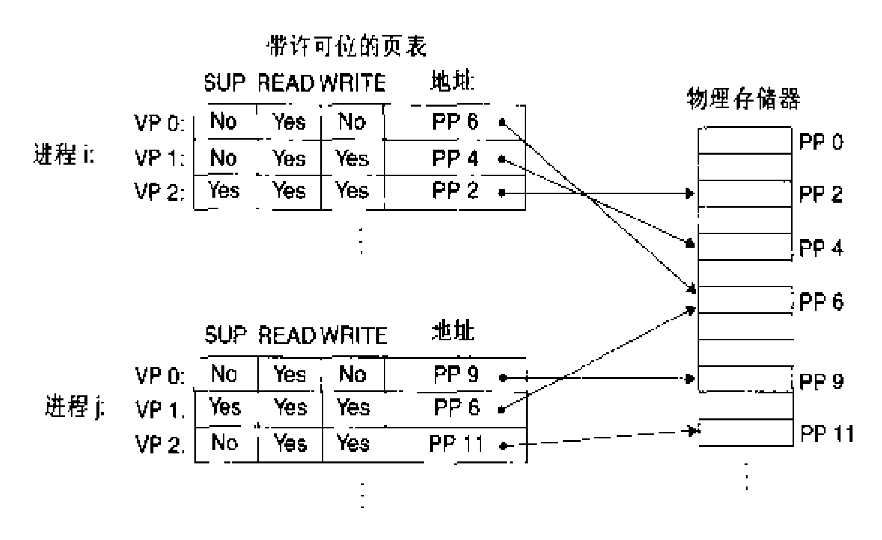


图 10.11 用虚拟存储器来提供页面级的存储器保护

在这个示例中，我们已经添加了三个许可位到每个 PTE。SUP 位表示进程是否必须运行在内核（超级用户）模式下才能访问该页。运行在内核模式中的进程可以访问任何页面，但是运行在用户模式中的进程只允许访问那些 SUP 为 0 的页面。READ 位和 WRITE 位控制对页面的读和写访问。例如，如果进程 i 运行在用户模式下，那么它有读 VP0 和读写 VP1 的权限。然而，不允许它访问 VP2。

如果一条指令违反了这些许可条件，那么 CPU 就触发一个一般保护故障，将控制传递给一个内核中的异常处理程序。Unix shell 典型地将这种异常报告为“段错误（segmentation fault)”。

10.6 地址翻译

这一节讲述的是地址翻译的基础知识。我们的目标是让你对硬件在支持虚拟存储器中的角色有正确的评价，并给你足够多的细节使得你可以亲手演示一些具体的示例。不过，要记住我们省略了大量的细节，尤其是和时钟相关的细节，虽然这些细节对硬件设计者来说是非常重要的，但是超出了我们讨论的范围。图 10.12 概括了我们在这节里将要使用的所有符号，供你参考。

地址翻译是一个 N 元素的虚拟地址空间（VAS）中的元素和一个 M 元素的物理地址空间（PAS）中元素之间的映射

$$MAP: VAS \rightarrow PAS \cup \emptyset$$

这里

MAP（A）= A' 如果虚拟地址 A 处的数据在 PAS 的物理地址 A'处。

= ∅ 如果虚拟地址 A 处的数据不在物理存储器中。

图 10.13 展示了 MMU 是如何利用页表来实现这种映射的。CPU 中的一个控制寄存器，*页表基址寄存器*（page table base register，PTBR）指向当前页表。n 位的虚拟地址包含两个部分：一个 p 位的 VPO（virtual page offset，*虚拟页面偏移*）和一个（n-p）位的 VPN（virtual page number，*虚拟页号*）。MMU 利用 VPN 来选择适当的 PTE。例如，VPN0 选择 PTE0，VPN1 选择 PTE1，以此类推。将页表条目中 PPN（physical page number，*物理页号*）和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。注意，因为物理和虚拟页面都是 P 字节的，所以 PPO（physical page offset，*物理页面偏移*）和 VPO 是相同的。

基本参数	
符号	描述
$N = 2^n$	虚拟地址空间中的地址数量
$M = 2^m$	物理地址空间中的地址数量
$P = 2^p$	页的大小（字节）

虚拟地址（VA）的组成部分	
符号	描述
VPO	虚拟页面偏移量（字节）
VPN	虚拟页号
TLBI	TLB 索引
TLBT	TLB 标记

物理地址（PA）的组成部分	
符号	描述
PPO	物理页面偏移量（字节）
PPN	物理页号
CO	缓冲块内的字节偏移量
CI	高速缓存索引
CT	高速缓存标记

图 10.12 地址翻译符号小结

图 10.14（a）展示了当出现页面命中时，CPU 硬件执行的步骤。

- *第一步*：处理器生成一个虚拟地址，并把它传送给 MMU。
- *第二步*：MMU 生成 PTE 地址，并从高速缓存/主存请求得到它。
- *第三步*：高速缓存/主存向 MMU 返回 PTE。
- *第四步*：MMU 构造物理地址，并把它传送给高速缓存/主存。
- *第五步*：高速缓存/主存返回所请求的数据字给处理器。

和页面命中不同的是，页面命中完全是由硬件来处理的，而处理缺页要求硬件和操作系统内核

协作完成，如图10.14（b）。

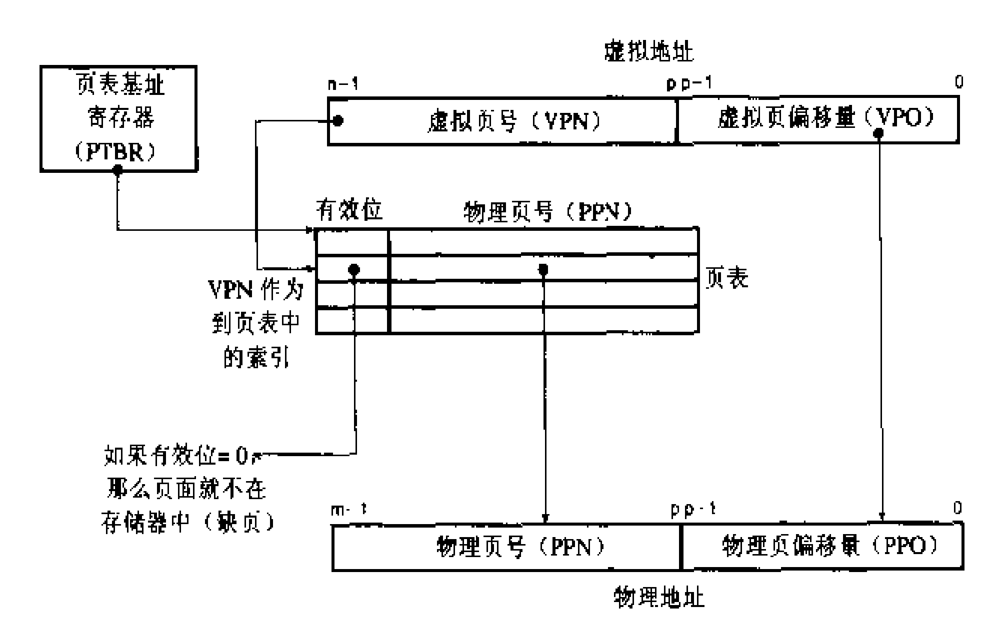
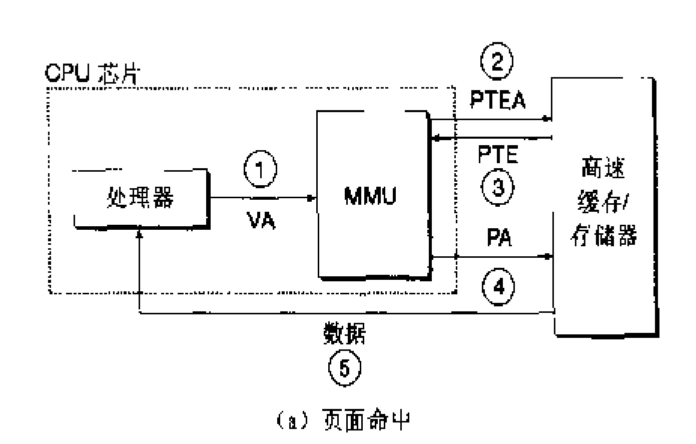
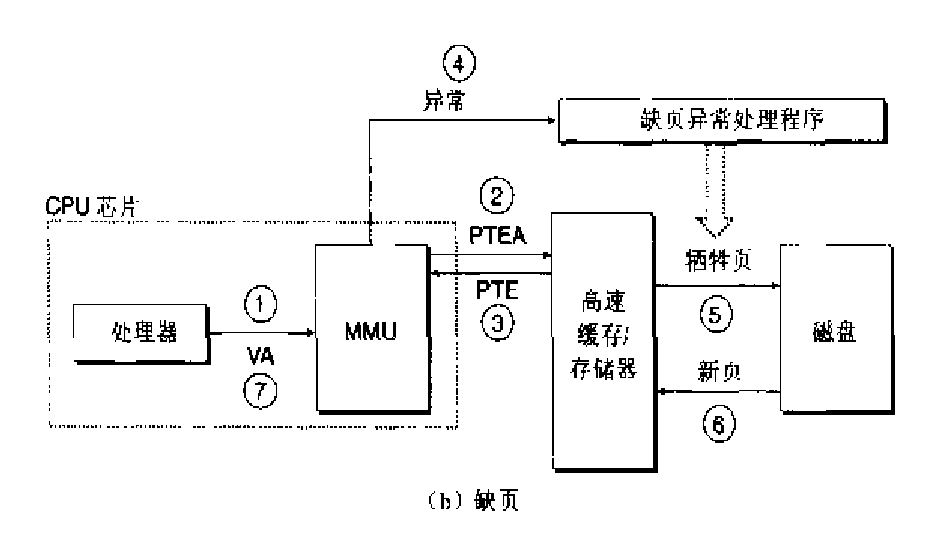


图10.13 使用页表的地址翻译



（a）页面命中



（b）缺页

图10.14 页面命中和缺页的操作视图

VA：虚拟地址；PTEA：页表条目地址；PTE：页表条目；PA：物理地址。

- 第一步到第三步：和图 10.14（a）中的第一步到第三步相同。
- 第四步：PTE 中的有效位是零，所以 MMU 触发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。
- 第五步：缺页处理程序确定出物理存储器中的牺牲页，如果这个页面已经被修改了，则把它页面换出到磁盘。
- 第六步：缺页处理程页面调入新的页面，并更新存储器中的 PTE。
- 第七步：缺页处理程序返回到原来的进程，驱使导致缺页的指令重新启动。CPU 将引起缺页的指令重新发送给 MMU。因为虚拟页面现在缓存在物理存储器中，所以就会命中，在 MMU 执行了图 10.14（b）中的步骤之后，主存就会将所请求字返回给处理器。

练习题 10.3

给定一个 32 位的虚拟地址空间和一个 24 位的物理地址，对于下面的页面大小 P，确定 VPN、VPO、PPN 和 PPO 中的位数：

P	# VPN 位	# VPO 位	# PPN 位	# PPO 位
1KB				
2KB				
4KB				
8KB				

10.6.1 结合高速缓存和虚拟存储器

在任何既使用虚拟存储器又使用 SRAM 缓存的系统中，都有应该使用虚拟地址还是使用物理地址来访问高速缓存的问题。尽管关于这个折中的详细讨论已经超出了我们的讨论范围，但是大多数系统是选择物理寻址的。使用物理寻址，多个进程同时在高速缓存中有存储块和共享来自相同虚拟页面的块成为很简单的事情。而且，高速缓存无需处理保护问题，因为访问权限的检查是地址翻译过程的一部分。

图 10.15 展示了一个物理寻址的高速缓存如何和虚拟存储器结合起来。主要的思路是地址翻译发生在高速缓存查找之前。注意页表条目可以缓存，就像其他的数据字一样。

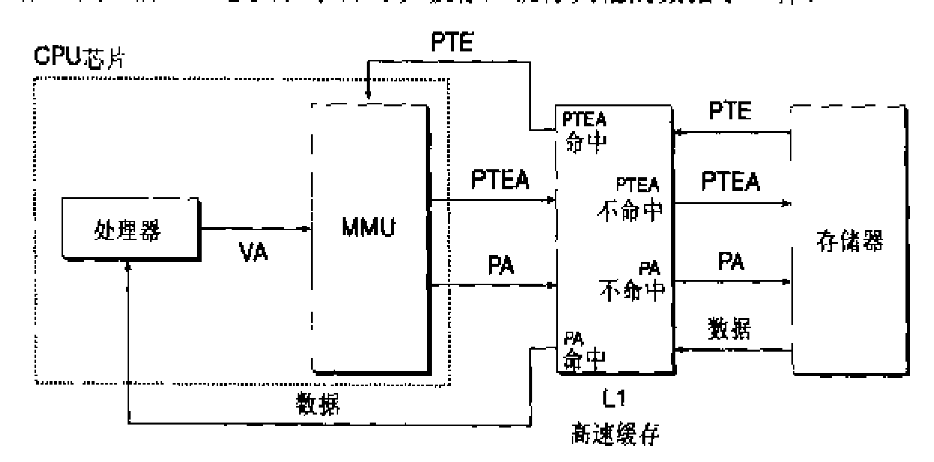


图 10.15 将虚拟存储器与一个物理寻址的高速缓存结合起来

VA：虚拟地址；PTEA：页表条目地址；PTE：页表条目；PA：物理地址。

10.6.2 利用TLB加速地址翻译

正如我们看到的，每次CPU产生一个虚拟地址，MMU就必须查阅一个PTE，以便将虚拟地址翻译为物理地址。在最糟糕的情况下，这会要求一次对存储器的额外的取数据，代价是几十到几百个周期。如果PTE碰巧缓存在L1中，那么开销就下降到1个或2个周期。然而，许多系统都试图消除即使是这样的开销，它们在MMU中包括了一个关于PTE的小的缓存，称为TLB（translation lookaside buffer，翻译后备缓冲器）。

TLB是一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单个PTE组成的块。TLB通常有高度的相联性。如图10.16所示，用于组选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号中提取出来的。如果TLB有$T=2^t$个组，那么TLB索引（TLBI）是由VPN的t个最低位组成的，而TLB标记（TLBT）是由VPN中剩余的位组成的。

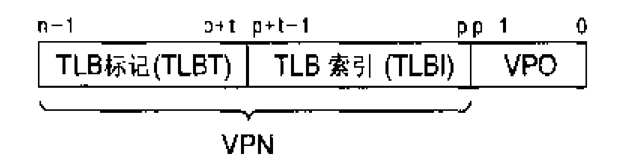
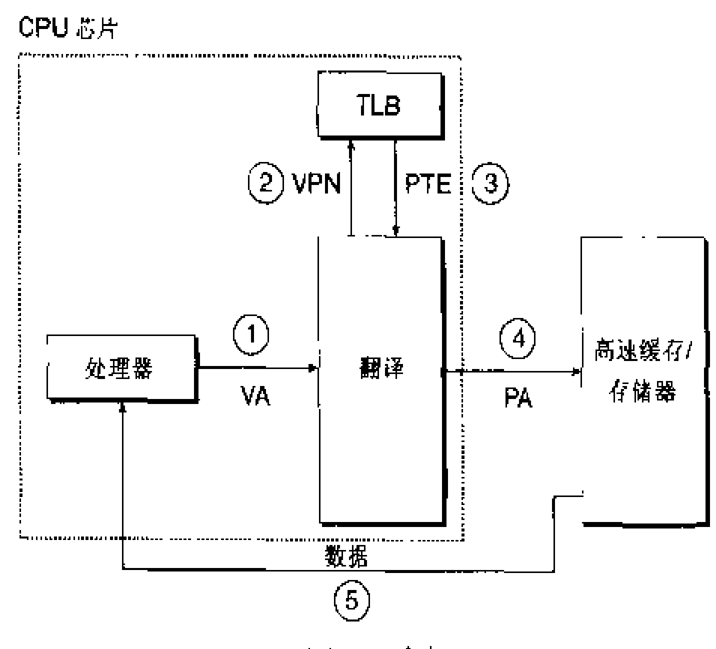


图10.16　一个用来访问TLB的虚拟地址的组成部分

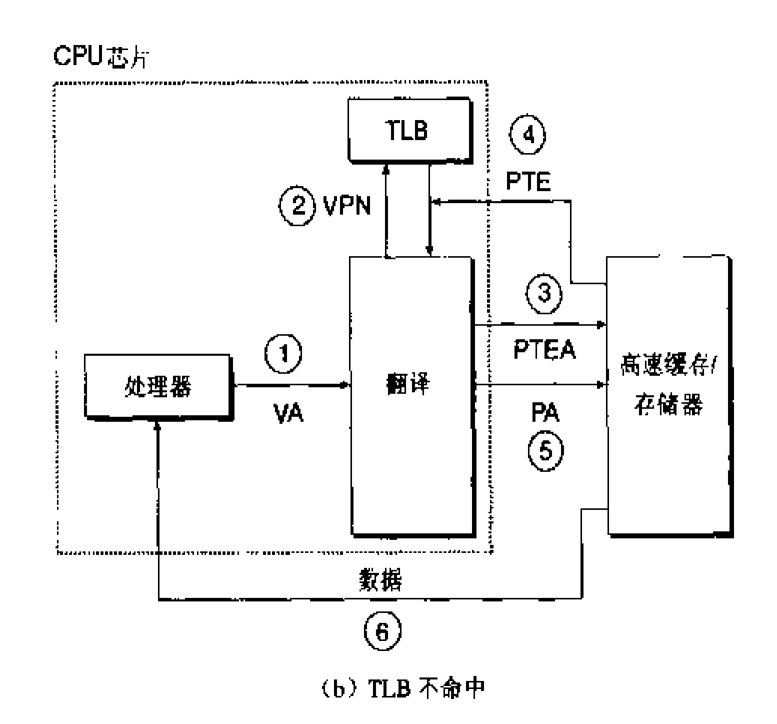
图10.17（a）展示了当TLB命中时（通常情况）所包括的步骤。这里的关键点是，所有的地址翻译步骤都是在MMU上执行的，因此非常快。

- 第一步：CPU产生一个虚拟地址
- 第二步和第三步：MMU从TLB中取出相应的PTE。
- 第四步：MMU将这个虚拟地址翻译成一个物理地址，并且将它发送到高速缓存/主存。
- 第五步：高速缓存/主存将所请求的数据字返回给CPU。

当TLB不命中时，MMU必须从L1缓存中取出相应的PTE，如图10.17（b）所示。新取出的PTE存放在TLB中，可能会覆盖一个已经存在的条目。



(a) TLB命中



(b) TLB 不命中

图 10.17 TLB 命中和不命中的操作视图

10.6.3 多级页表

到目前为止，我们一直假设系统只用一个单独的页表来进行地址翻译。但是如果我们有一个 32 位的地址空间、4KB 的页面和一个 4 字节的 PTE，那么我们总是需要一个 4MB 的页表驻留在存储器中，即使应用所引用的只是虚拟地址空间中很小的一部分。对于地址空间为 64 位的系统来说，问题将变得更复杂。

用来压缩页表的常用方法是使用层次结构的页表。我们使用一个具体的示例来加深你对这种方法的理解。假设 32 位虚拟地址空间被分为 4KB 的页，而每个页表条目都是 4 字节。还假设在这一时刻，虚拟地址空间有如下形式：存储器的头 2K 个页面分配给了代码和数据，接下来的 6K 个页面还未分配，再接下来的 1 023 个页面也未分配，接下来的 1 个页面分配给了用户栈。图 10.18 展示了我们如何为这个虚拟地址空间构造一个两级的页表层次结构。

一级页表中的每个 PTE 负责映射虚拟地址空间中一个 4MB 的组块（chunk），这里每个组块都是由 1 024 个连续的页面组成的。比如，PTE 0 映射第一个组块，PTE 1 映射接下来的一组块，以此类推。假设地址空间是 4GB，1 024 个 PTE 已经足够覆盖整个空间了。

如果组块 i 中的每个页面都未被分配，那么一级 PTE i 就为空。例如，图 10.18 中，组块 2～7 是未被分配的。然而，如果在组块 i 中至少有一个页是分配了的，那么一级 PTE i 就指向一个二级页表的基址。例如，如图 10.18 所示，组块 0、1 和 8 的所有或者部分已被分配，所以它们的一级 PTE 就指向二级页表。

二级页表中的每个 PTE 都负责映射一个 4KB 的虚拟存储器页面，就像我们查看只有一级的页表一样。注意，使用 4 字节的 PTE，每个一级和二级页表都是 4KB 字节，这刚好和一个页面的大小是一样的。

这种方法从两个方面减少了存储器要求。第一，如果一级页表中的一个 PTE 是空的，那么相应的二级页表就根本不会存在。这表现出一种巨大的潜在节约，因为对于一个典型的程序，4GB 的虚

拟地址空间的大部分都会是未分配的。第二，只有一级页表才需要总是在主存中。虚拟存储器系统可以在需要时创建，并页面调入或调出二级页表，这就减少了主存的压力。只有最经常使用的二级页表才需要缓存在主存中。

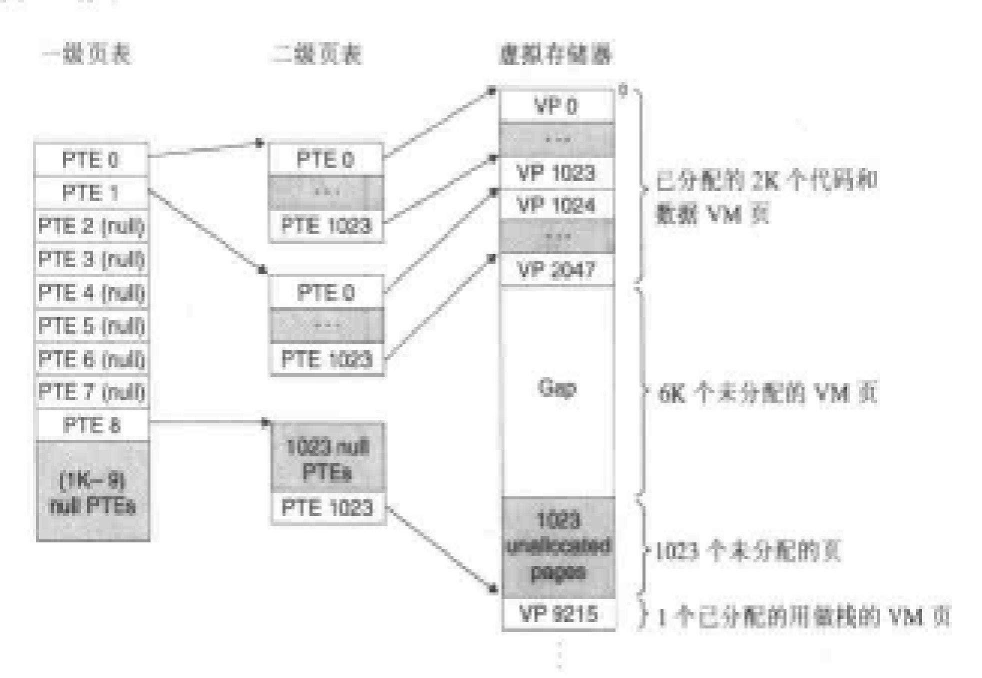


图 10.18　一个两级页表层次结构

注意地址是从上往下增加的。

图 10.19 描述了使用 k 级页表层次结构的地址翻译。虚拟地址被划分成为 k 个 VPN 和 1 个 VPO。每个 VPN i 都是一个到第 i 级页表的索引，其中 $1 \le i \le k$。第 j 级页表中的每个 PTE，$1 \le j \le k-1$，都指向第 j+1 级的某个页表的基址。第 k 级页表中的每个 PTE 都包含某个物理页面的 PPN，或者一个磁盘块的地址。为了构造物理地址，在能够确定 PPN 之前，MMU 必须访问 k 个 PTE。对于只有一级的页表结构，PPO 和 VPO 是相同的。

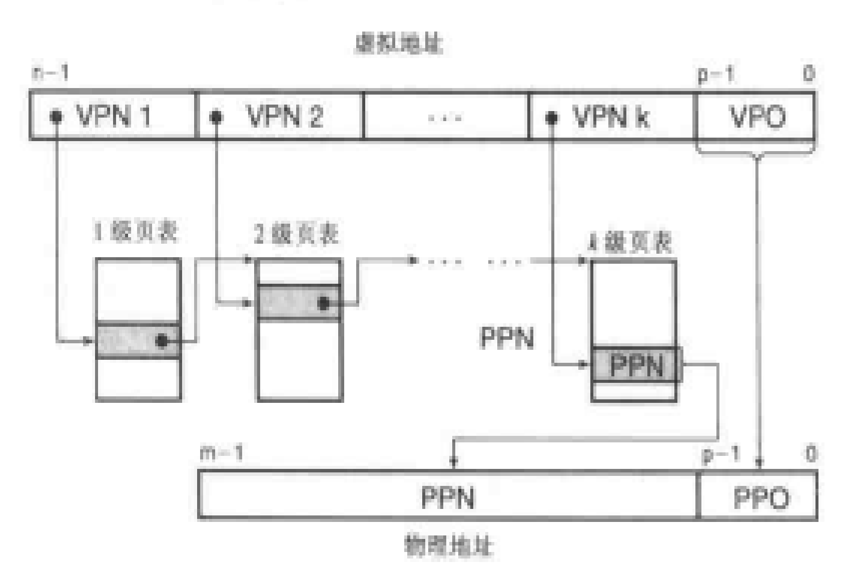


图 10.19　使用 k 级页表的地址翻译

访问 k 个 PTE，第一眼看上去昂贵而不切实际。然而，这里 TLB 能够起作用，正是通过将页表中不同层次上的 PTE 缓存起来。实际中，带多级页表的地址翻译并不比单级页表慢很多。

10.6.4 综合：端到端的地址翻译

在这一节里，我们通过一个具体的端到端的地址翻译示例，来综合一下我们刚学过的这些内容，这个示例运行在有一个 TLB 和 L1 d-cache 的小系统上。为了保证可管理性，我们做出如下假设：

- 存储器是按字节寻址的。
- 存储器访问是针对 1 字节的字的（不是 4 字节的字）。
- 虚拟地址是 14 位长的（n=14）。
- 物理地址是 12 位长的（m=12）。
- 页面大小是 64 字节（P=64）。
- TLB 是四路组相联的，总共有 16 个条目。
- L1 d-cache 是物理寻址、直接映射的，行大小为 4 字节，而总共有 16 个组。

图 10.20 展示了虚拟地址和物理地址的格式。因为每个页面是 2^6=64 字节，所以虚拟地址和物理地址的低 6 位分别作为 VPO 和 PPO。虚拟地址的高 8 位作为 VPN。物理地址的高 6 位作为 PPN。

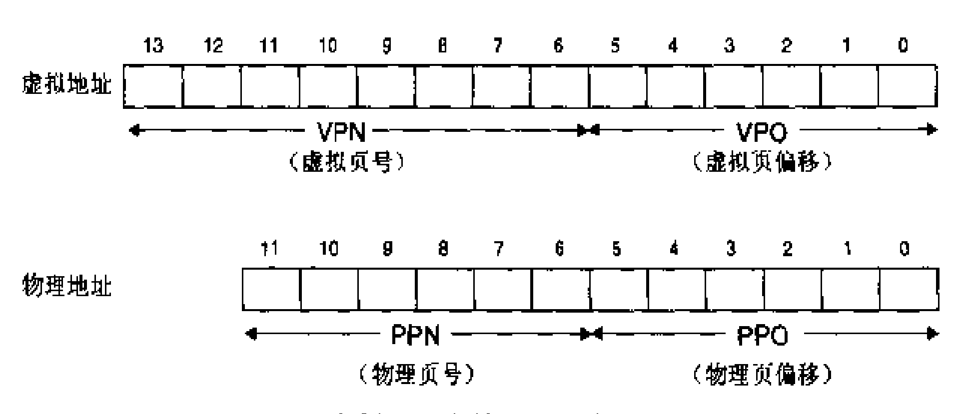
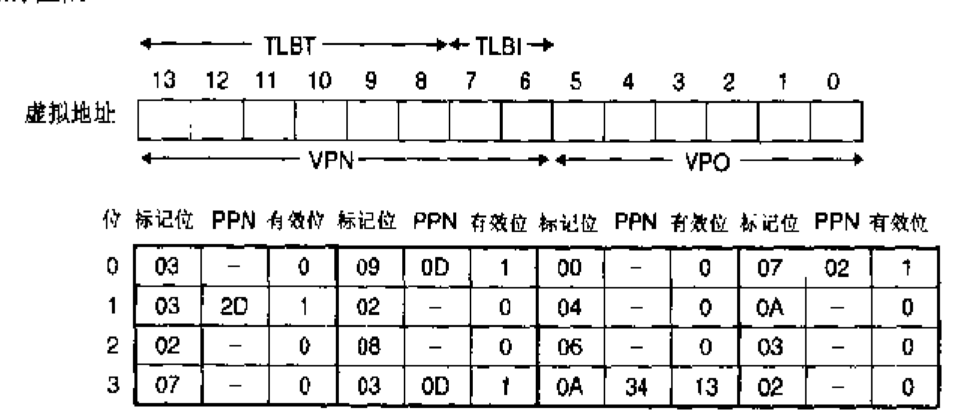


图 10.20 小存储器系统的寻址

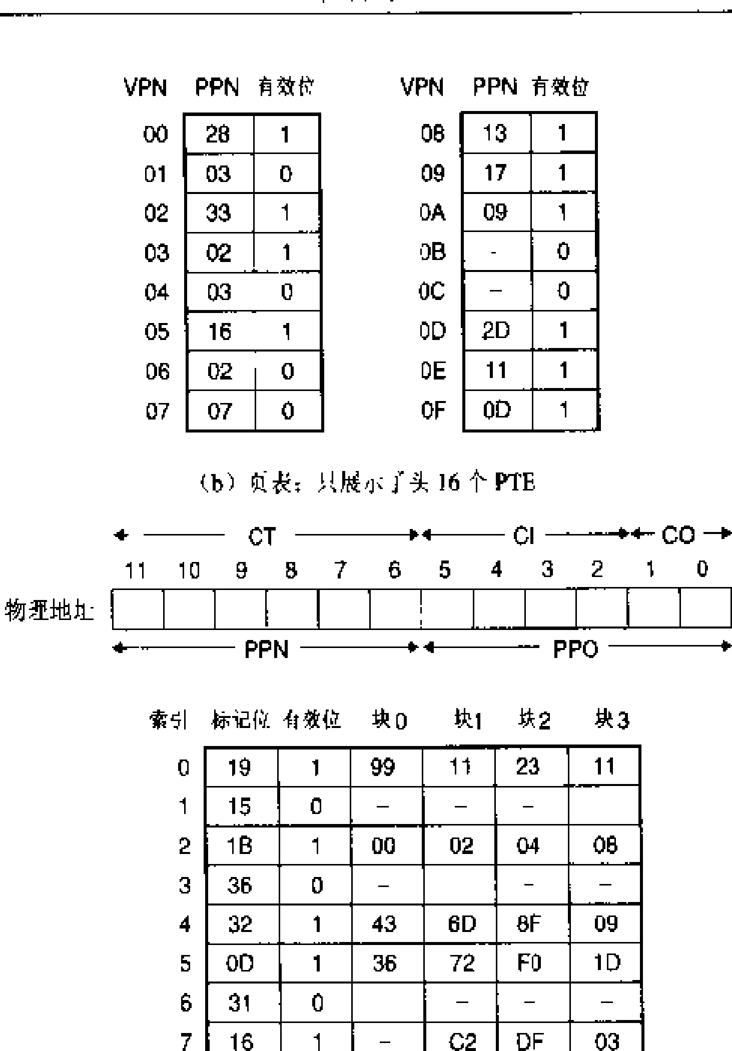
假设 14 位的虚拟地址（n=14），12 位的物理地址（m=12）和 64 字节的页面（P=64）。

图 10.21 展示了我们小存储器系统的一个快照，包括 TLB（a）、页表的一部分（b）和 L1 高速缓存（c）。在 TLB 和高速缓存的图表上面，我们还展示了访问这些设备的硬件是如何划分虚拟地址和物理地址的位的。



组	标记位	PPN	有效位	标记位	PPN	有效位	标记位	PPN	有效位	标记位	PPN	有效位
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

（a）TLB：四组，16 个条目，四路组相联



VPN	PPN	有效位	VPN	PPN	有效位
00	28	1	08	13	1
01	03	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	-	0
04	03	0	0C	–	0
05	16	1	0D	2D	1
06	02	0	0E	11	1
07	07	0	0F	0D	1

（b）页表：只展示了头 16 个 PTE

索引	标记位	有效位	块0	块1	块2	块3
0	19	1	99	11	23	11
1	15	0	–	–	–	
2	1B	1	00	02	04	08
3	36	0	–		–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0		–	–	–
7	16	1	–	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

（c）缓存：16 个组，四字节的块，直接映射

图 10.21 小存储器系统的 TLB、页表以及缓存

TLB、页表和缓存中所有的值都是十六进制表示的。

- TLB。TLB 是利用 VPN 的位进行虚拟寻址的。因为 TLB 有四个组，所以 VPN 的低两位就作为组索引（TLBI）。VPN 中剩下的高 6 位作为标记（TLBT），用来区别可能映射到同一个 TLB 组的不同的 VPN。
- 页表。这个页表是一个单级设计，一共有 2^8=256 个页表条目（PTE）。然而，我们只对这些条目中的开头 16 个感兴趣。为了方便，我们用索引它的 VPN 来标识每个 PTE；但是要记住这些 VPN 并不是页表的一部分，也不储存在存储器中。另外，注意每个无效 PTE 的 PPN

都用一个破折号来表示，以巩固一个概念：无论刚好这里存储的是什么位值，都是没有任何意义的。

- 缓存。直接映射的缓存是通过物理地址中的字段来寻址的。因为每个块都是 4 字节，所以物理地址的低 2 位作为块偏移（CO）。因为有 16 组，所以接下来的 4 位就用来表示组索引（CI）。剩下的 6 位作为标记（CT）。

给定了这种初始化设定，让我们来看看当 CPU 执行一条读地址 0x03d4 处字节的加载指令时，会发生什么。（回想一下我们假定 CPU 读取 1 字节的字，而不是 4 字节的字。）为了开始这种手工的模拟，我们发现写下虚拟地址的位表示，标识出我们会需要的各种字段，并确定它们的 16 进制值，是非常有帮助的。当硬件解码地址时，它也执行相似的任务。

	TLBT						TLBI							
	0x03						0x03							
位位置	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

开始时，MMU 从虚拟地址中抽取出 VPN（0x0F），并且检查 TLB 看它是否因为前面的某个存储器引用，缓存了 PTE 0x0F 的一个拷贝。TLB 从 VPN 中抽取出 TLB 索引（0x03）和 TLB 标记（0x3），组 0x3 的第二个条目中有效位匹配，所以命中，然后将缓存的 PPN（0x0D）返回给 MMU。

如果 TLB 不命中，那么 MMU 就需要从主存中取出相应的 PTE。然而，在这里的情况中，我们很幸运，TLB 会命中。现在，MMU 有了形成物理地址所需要的所有东西。它通过将来自 PTE 的 PPN（0x0D）和来自虚拟地址的 VPO（0x14）连接起来，这就形成了物理地址（0x354）。

接下来，MMU 发送物理地址给缓存，缓存从物理地址中抽取出缓存偏移 CO（0x0）、缓存组索引 CI（0x5）以及缓存标记 CT（0x0D）。

	CT						CI				CO	
	0x0d						0x05				0x0	
位位置	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

因为组 0x5 中的标记与 CT 相匹配，所以缓存检测到一个命中，读出在偏移量 CO 处的数据字节（0x36），并将它返回给 MMU，随后 MMU 将它传递回 CPU。

翻译过程的其他路径也是可能的。例如，如果 TLB 不命中，那么 MMU 必须从页表中的 PTE 中取出 PPN。如果得到的 PTE 是无效的，那么就产生一个缺页，内核必须调入合适的页面，重新运行这条加载指令。另一种可能性是 PTE 是有效的，但是所需要的存储器块在缓存中不命中。

练习题 10.4

说明 10.6.4 节中的示例存储器系统是如何将一个虚拟地址翻译成一个物理地址，以及访问缓存的。对于给定的虚拟地址，指明访问的 TLB 条目、物理地址和返回的缓存字节值。指出是否发

生了 TLB 不命中，是否发生了缺页，以及是否发生了缓存不命中。如果是缓存不命中，在“返回的缓存字节”栏中输入“-”。如果有缺页，则在“PPN”一栏中输入“-”，并且空着 C 部分和 D 部分。

虚拟地址： 0x03d7

A. 虚拟地址格式

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. 地址翻译

参 数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中？（Y/N）	
缺页？（Y/N）	
PPN	

C. 物理地址格式

11	10	9	8	7	6	5	4	3	2	1	0

D. 物理存储器引用

参 数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中？（Y/N）	
返回的缓存字节	

10.7 案例研究：Pentium/Linux 存储器系统

我们以一个实际系统的案例研究来概括我们对缓存和虚拟存储器的讨论，选用的系统是 Pentium 类的系统，运行的是 Linux。图 10.22 给出了 Pentium 存储器系统的重要部分。Pentium 系统有一个 32 位（4GB）的地址空间。处理器组件（processor package）包括 CPU 芯片、一个统一的 L2 高速缓存和一个连接它们的高速缓存总线（背板总线）。CPU 芯片适当地包含了四个不同的缓存：一个指令 TLB、数据 TLB、L1 i-cache 以及 L1 d-cache。TLB 是虚拟寻址的。L1 和 L2 缓存是物理寻址的。Pentium 中的所有缓存（包括 TLB）都是四路组相联的。

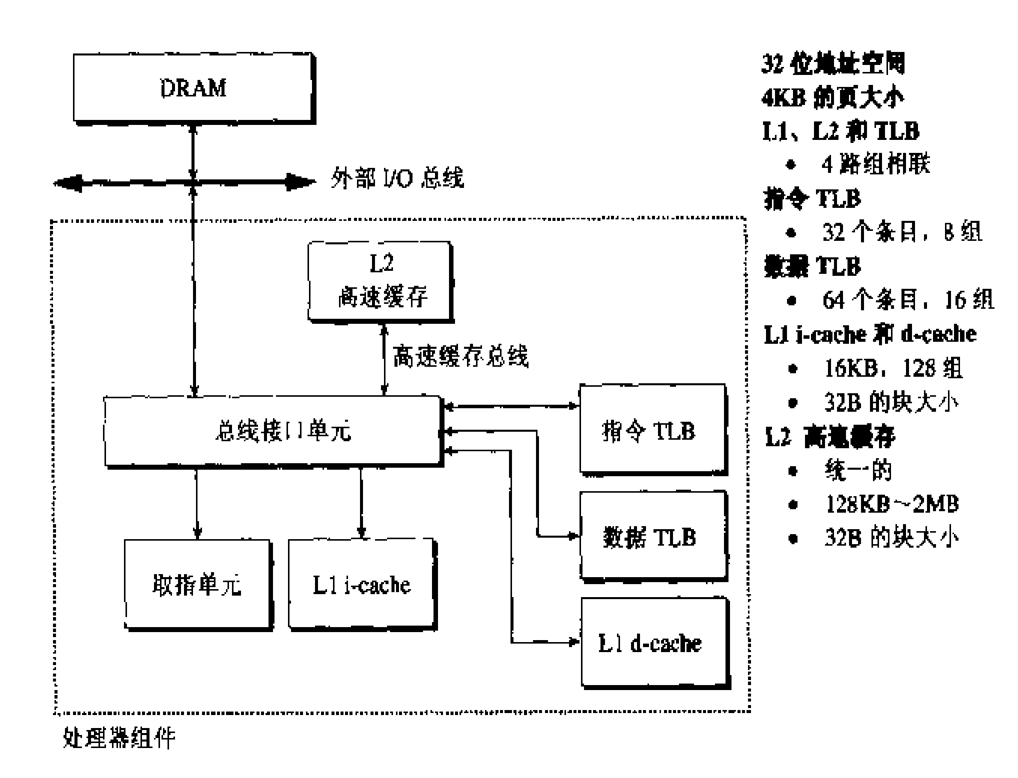


图 10.22 Pentium 存储器系统

TLB 缓存 32 位的页表条目。指令 TLB 缓存取指单元产生的虚拟地址的 PTE。数据 TLB 缓存数据的虚拟地址的 PTE。指令 TLB 有 32 个条目。数据 TLB 有 64 个条目。页面大小可以在启动时配置成 4KB 或者 4MB。运行在 Pentium 上的 Linux 使用 4KB 的页面。

L1 和 L2 高速缓存的块大小为 32 字节。每个 L1 高速缓存的大小是 16KB，有 128 个组，其中每个组都包含 4 行。L2 高速缓存的大小可以在最小值 128KB 到最大值 2MB 之间变化。典型的大小是 512KB。

10.7.1 Pentium 地址翻译

这一节讨论 Pentium 系统上的地址翻译过程。图 10.23 描述了整个过程，从 CPU 产生虚拟地址时，直到数据字从存储器到达，以供你参考。

旁注：优化地址翻译

在我们对地址翻译的讨论中，我们已经描述了一个顺序的两个步骤的过程，就是 MMU 将虚拟地址翻译成物理地址，然后将物理地址传送到 L1 高速缓存。然而，实际的硬件实现使用了一个灵巧的技巧，允许这些步骤部分重叠，因此也就加速了对 L1 高速缓存的访问。

例如，带 4KB 页面的 Pentium 系统上的一个虚拟地址有 12 位的 VPO，并且这些位和相应物理地址中的 PPO 的 12 位是相同的。因为四路组相联的、物理寻址的 L1 高速缓存有 128 个组和 32 字节的缓存块，每个物理地址有 5 个（$\log_2 32$）缓存偏移位和 7 个（$\log_2 128$）索引位。这 12 个位恰好符合虚拟地址的 VPO 部分，这绝不是偶然！当 CPU 需要翻译一个虚拟地址时，它就发送 VPN 到 MMU，发送 VPO 到高速 L1 缓存。当 MMU 向 TLB 请求一个页表条目时，L1 高速缓存正忙着利用 VPO 位查找相应的组，并读出这个组里的四个标记和相应的数据字。当 MMU 从 TLB 得到 PPN 时，缓存已经准备好试着把这个 PPN 与这四个标记中的一个进行匹配了。

这就引发了下面的问题让你思考：如果 Intel 的工程师在未来的系统中想增加 L1 高速缓存的大小，并且仍然能够使用这种技巧，那么他们有什么样的选择呢？

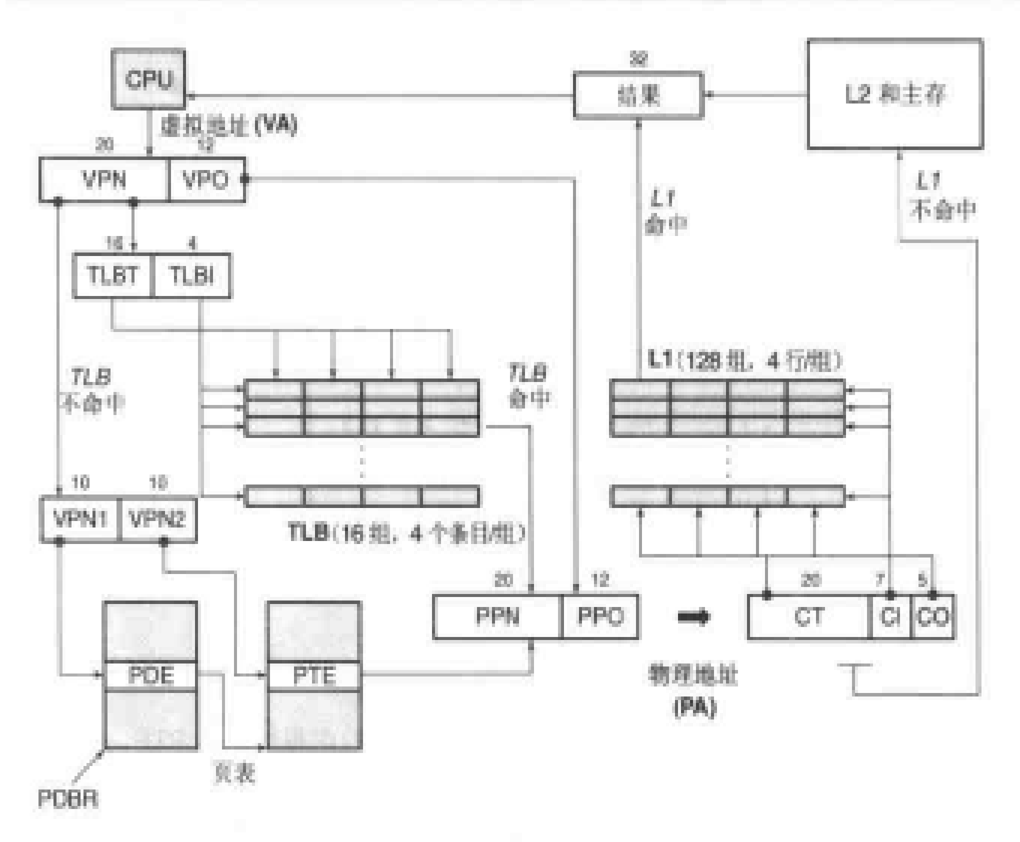


图 10.23 Pentium 地址翻译的概况

Pentium 页表

每个 Pentium 系统都使用如图 10.24 所示的两级页表。第一级页表，叫做页面目录（page directory），包含 1024 个 32 位的 PDE（page directory entry，页面目录条目），其中每一个条目都指向 1024 个 2 级页表中的一个。每个页表包含 1024 个 32 位的 PTE（页表条目），其中每个都指向物理存储器或者磁盘上的一个页面。

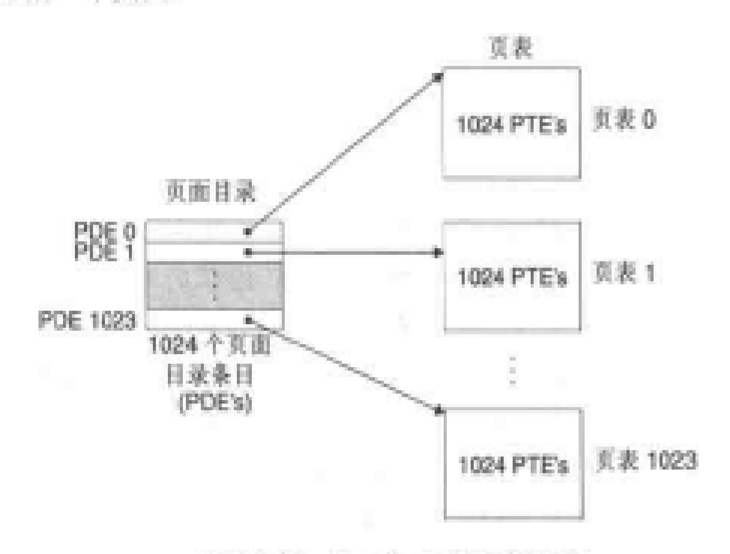


图 10.24 Pentium 的多级页表

每个进程都有一个惟一的页面目录和页表集合。当一个 Linux 进程正在运行时，尽管 Pentium 的体系结构允许页表换进换出，但是页表目录和与已分配页面相关的页表都是常驻存储器的。页面目录基址寄存器（page directory base register，PDBR）指向页表目录的起始位置。

图 10.25（a）展示了 PDE 的格式。当 P=1 时（Linux 中总是这样的），地址字段中包含一个 20 位的物理页号，指向相应的页表的起始位置。注意，这要求页表要 4KB 对齐。

图 10.25（b）展示了 PTE 的格式。当 P=1 时，地址字段包含一个 20 位的物理页号，指向物理存储器中某个页的基址。同样，这也要求物理页要 4KB 对齐。

31–12	11–9	8	7	6	5	4	3	2	1	0
页表物理基地址	未用	G	PS		A	CD	WT	U/S	R/W	P=1

字段	描　述
P	页表存在于物理存储器中（1）或者不存在（0）
R/W	只读或者读-写访问许可
U/S	用户或者超级用户模式（内核模式）访问许可
WT	对这个页表的直写或者写回缓存策略
CD	缓存禁止（1）或者启用（0）
A	这个页被访问过吗？（在读写时由MMU设置，由软件清除）
PS	页面大小为4KB（0）或者4MB（1）
G	全局页面（在任务切换时，不会从TLB中驱除掉）
PT 基地址	物理页表地址的最高 20 位

（a）页面目录条目（PDE）

31–12	11–9	8	7	6	5	4	3	2	1	0
页面物理基地址	未用	G	0	D	A	CD	WT	U/S	R/W	P=1

31–1	0
操作系统可用的（在二级存储中的页面位置）	P=0

字段	描　述
P	页表存在于物理存储器中（1）或者不存在（0）
R/W	只读或者读/写访问许可
U/S	用户或者超级用户模式（内核模式）访问许可
WT	对这个页表的直写或者写回缓存策略
CD	缓存禁止（1）或者启用（0）
A	引用位（由MMU在读写时设置，由软件清除）
D	修改位（由MMU在写时设置，由软件清除）
G	全局页面（在任务切换时不会从TLB中驱除掉）
页面基地址	物理页表地址的最高 20 位

（b）页表条目（PTE）

图 10.25 Pentium 页面目录条目（PDE）和页表条目（PTE）的格式

PTE 有两个许可位，用来控制对这个页面的访问。R/W 位确定这个页面的内容是可读/可写的，还是只读的。U/S 位，确定是否可以在用户模式下访问这个页面，这就保护了操作系统内核中的代码和数据不受用户程序的影响。

就像 MMU 翻译每个虚拟地址一样，它也会更新两个其他的位，内核的缺页处理程序可能会使用这两位。每次访问一个页面时，MMU 就设置 A 位，也叫做引用位（reference bit）。内核可以用引用位来实现它的页面替换算法。每次写页面时，MMU 就设置 D 位，也叫做修改位（dirty bit）。一个已经被修改了的页面有时也叫做*修改页面*（dirty page）。修改位告诉内核在它拷入一个替代页面时，是否必须写回牺牲页面。内核可以调用一个特殊的内核模式指令来清除引用或者修改位。

旁注：执行许可和缓冲区溢出攻击

注意，Pentium 页表条目缺少一个执行许可位，用来控制一个页面的内容是否可以被执行。缓冲区溢出攻击利用了这个疏漏，在用户栈上直接加载和运行代码（3.13 节）。如果有这样一个执行位，那么内核就可以通过限制对只读代码段的执行权限，来消除这种攻击的威胁了。

Pentium 页表翻译

图 10.26 展示了 Pentium MMU 如何使用两级页表，将一个虚拟地址翻译成物理地址。20 位的 VPN 被分成 2 个 10 位的块。VPN1 在 PDBR 指向的页目录中索引一个 PDE。PDE 中的地址指向的某个页表的基址被 VPN2 索引。被 VPN2 索引的 PTE 中的 PPN 和 VPO 连接起来形成了物理地址。

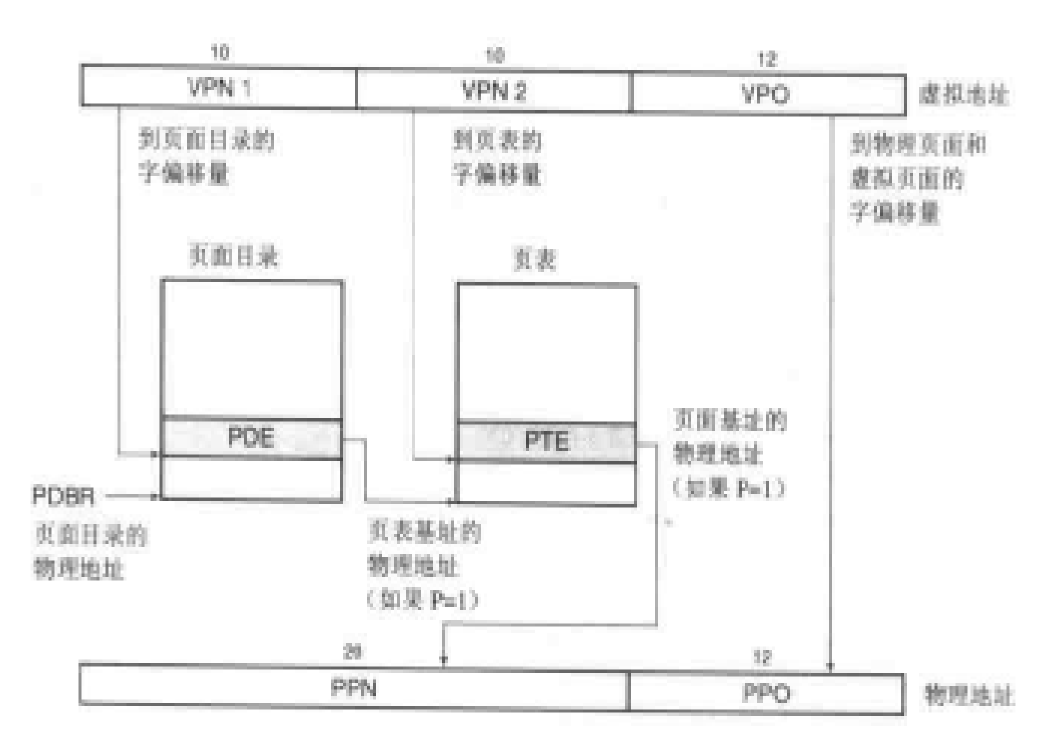


图 10.26 Pentium 页表翻译

Pentium TLB 翻译

图 10.27 描绘了 Pentium 系统中 TLB 翻译的过程。如果 PTE 被缓存在 TLBI 索引的组里（TLB

命中），那么就从这个缓存的 PTE 中抽取出 PPN，并把这个 PPN 和 VPO 连接起来形成物理地址。如果没有缓存 PTE，但是缓存了 PDE（部分 TLB 命中），那么 MMU 必须在它形成物理地址之前，从存储器中提取相应的 PTE。最后，如果 PDE 和 PTE 都没有被缓存（TLB 不命中），那么 MMU 必须从存储器中取出 PDE 和 PTE，以形成物理地址。

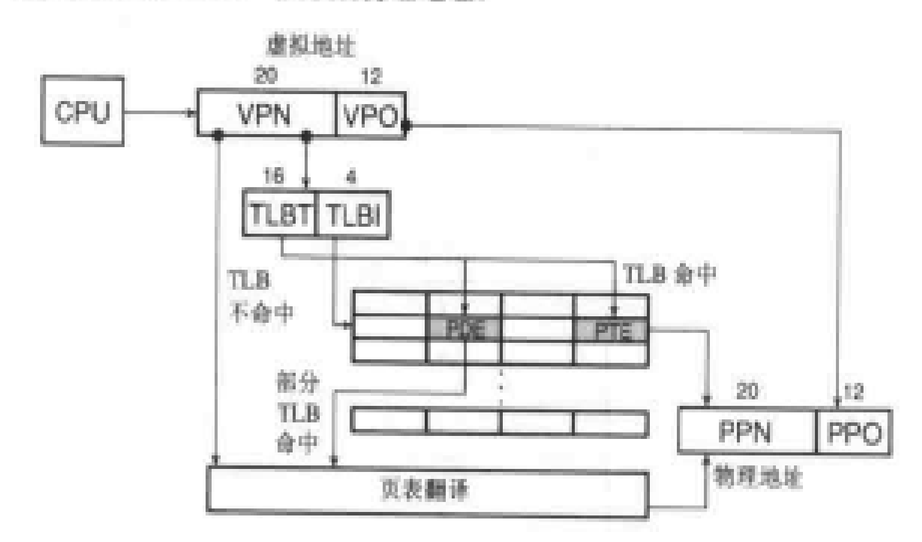


图 10.27 Pentium TLB 翻译

10.7.2 Linux 虚拟存储器系统

一个虚拟存储器系统要求硬件和内核软件之间的紧密协作，然而对此完整的阐释超出了我们讨论的范围，在这一小节中我们的目标是对 Linux 的虚拟存储器系统做一个描述，使你能够大致了解一个实际的操作系统是如何组织虚拟存储器，以及如何处理缺页的。

Linux 为每个进程维持了一个单独的虚拟地址空间，形式如图 10.28 所示。我们已经多次看到过这幅图了，包括它那些熟悉的代码、数据、堆、共享库以及栈段。既然我们已理解地址翻译，我们就能够填入更多的关于内核虚拟存储器的细节了，这部分虚拟存储器位于地址 0xc0000000 之上。

内核虚拟存储器包含内核中的代码和数据。内核虚拟存储器的某些区域被映射到所有进程共享的物理页面。例如，每个进程共享内核的代码和全局数据结构。有趣的是，Linux 也将一组连续的虚拟页面（大小等于系统中 DRAM 的总量）映射到相应的一组连续的物理页面。这就为内核提供了一种便利的方法，来访问物理存储器中任何特定的位置，例如，当它需要在一些设备上执行存储器映射的 I/O 操作时，而这些设备被映射到特定的物理存储器位置。

内核虚拟存储器的其他区域包含每个进程都不相同的数据。示例包括页表、内核在进程的上下文中执行代码时使用的栈，以及记录虚拟地址空间当前组织的各种数据结构。

Linux 虚拟存储器区域

Linux 将虚拟存储器组织成一些区域（也叫做段）的集合。一个区域（area）就是已经存在着的（已分配的）虚拟存储器的连续组块（chunk），这些虚拟存储器的页面是以某种方式相关联的。例如，代码段、数据段、堆、共享库段，以及用户栈都是不同的区域。每个存在的虚拟页面都保存在某个区域中，而不属于某个区域的虚拟页是不存在的，并且不能被进程引用。区域的概念很重要，因为它允许虚拟地址空间有间隙。内核并不记录那些不存在的虚拟页，而这样的页面也不占用存储

器、磁盘或者内核本身中的任何额外资源。

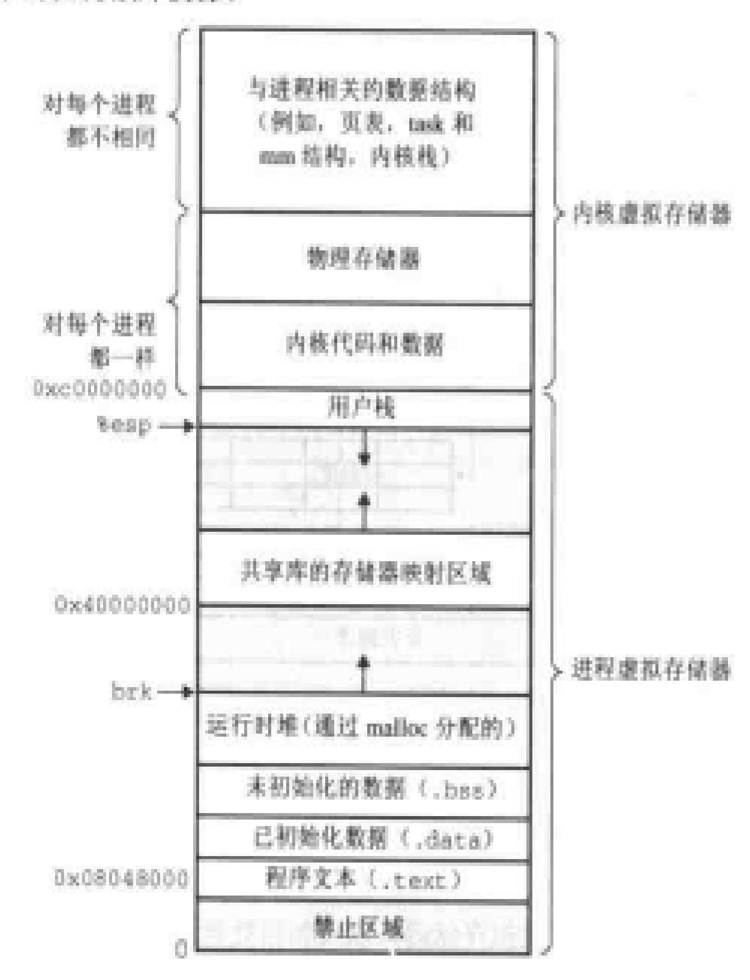


图 10.28 一个 Linux 进程的虚拟存储器

图 10.29 强调了记录一个进程中虚拟存储器区域的内核数据结构。内核在系统中为每个进程维护一个单独的任务结构（源代码中的 task_struct）。任务结构中的元素包含或者指向内核运行该进程所需要的所有信息（例如，PID、指向用户栈的指针、可执行目标文件的名字，以及程序计数器）。

task_struct 中的一个条目指向 mm_struct，它描述了虚拟存储器的当前状态。我们感兴趣的两个字段是 pgd 和 mmap，其中 pgd 指向页面目录表的基址，而 mmap 指向一个 vm_area_structs（区域结构）的链表，其中每个 vm_area_structs 都描述了当前虚拟地址空间的一个区域（area）。当内核运行这个进程时，它就将 pgd 存放在 PDBR 控制寄存器中。

为了我们的目的，一个具体区域的区域结构（vm_area_struct）包含下面的字段：

- vm_start：指向这个区域的起始处。
- vm_end：指向这个区域的结束处。
- vm_port：描述这个区域内包含的所有页面的读写许可权限。
- vm_flags：描述这个区域内的页面是否是与其他进程共享的，还是这个进程私有的（还描述了其他一些信息）。
- vm_next：指向链表中下一个区域结构。

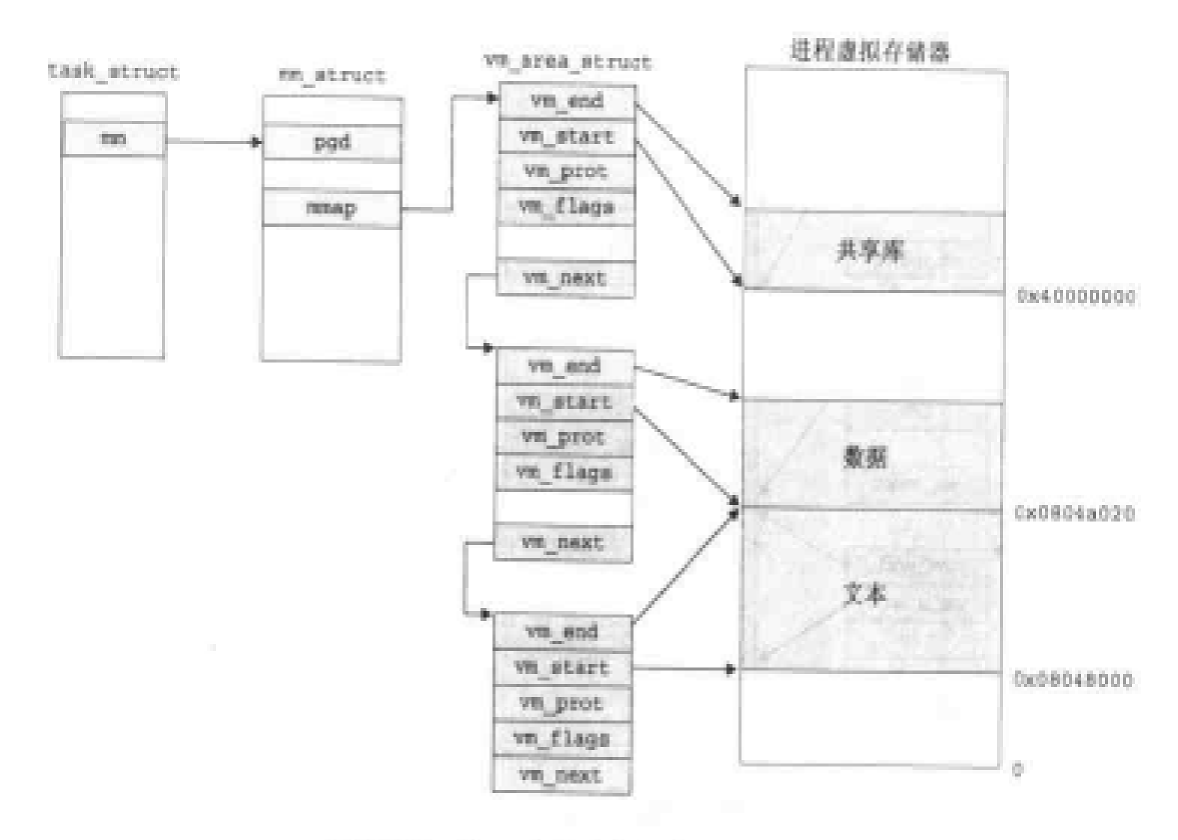


图 10.29 Linux 是如何组织虚拟存储器的

Linux 缺页异常处理

假设 MMU 在试图翻译某个虚拟地址 A 时，触发了一个缺页。这个异常导致控制转移到内核的缺页处理程序，处理程序随后就执行下面的步骤：

1. 虚拟地址 A 是合法的吗？换句话说，A 在某个区域结构（vm_area_struct）定义的区域内吗？为了回答这个问题，缺页处理程序搜索区域结构的链表，把 A 和每个区域结构中的 vm_start 和 vm_end 做比较。如果这个指令是不合法的，那么缺页处理程序就触发一个段错误，从而终止这个进程。这个情况在图 10.30 中标识为“1”。

因为一个进程可以创建任意数量的新虚拟存储器区域（使用在下一节中描述的 mmap 函数），所以顺序搜索区域结构的链表花销可能会很大。因此在实际中，使用某些我们没有显示出来的字段，Linux 在链表中添加了一棵树，并在这棵树上进行查找。

2. 试图进行的对存储器的访问是否合法？换句话说，进程是否有读或者写这个区域内页面的权限？例如，这个缺页是不是由一条试图对这个代码段里的只读页面进行写操作的存储指令造成的？这个缺页是不是因为一个运行在用户模式中的进程试图从内核虚拟存储器中读取字造成的？如果试图进行的访问是不合法的，那么缺页处理程序会触发一个保护异常，从而终止这个进程。这种情况在图 10.30 中标识为“2”。

3. 此刻，内核知道了这个缺页是由于对合法的虚拟地址进行合法的操作造成的。它选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面，并更新页表，从而处理这次缺页。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令将再次发送 A 到 MMU。这一回，MMU 就能正常地翻译 A，而不会再产生一个缺页中断了。

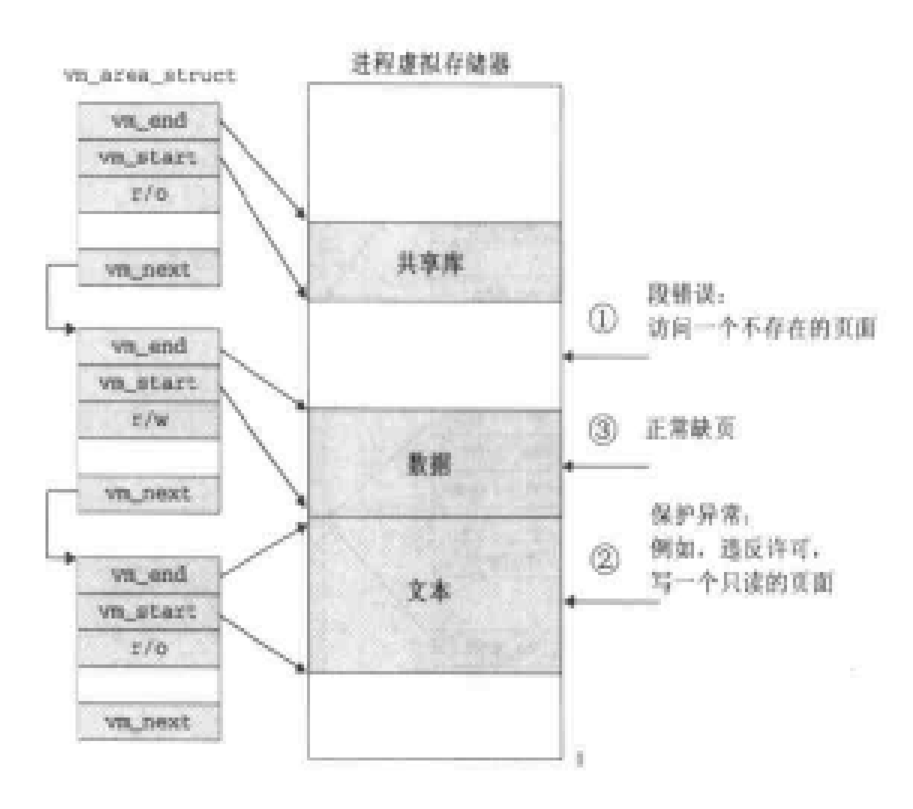


图10.30　Linux缺页处理

10.8　存储器映射

Linux（以及其他一些形式的Unix）通过将一个虚拟存储器区域与一个磁盘上的对象（object）关联起来，以初始化这个虚拟存储器区域的内容，这个过程称为*存储器映射*（memory mapping）。虚拟存储器区域可以映射到两种类型的对象：

1. *Unix文件系统中的普通文件*：一个区域可以映射到一个普通磁盘文件（例如一个可执行目标文件）的连续部分。文件被分成页面大小的片，每一片包含一个虚拟页面的初始内容。因为按需进行页面调度，所以这些虚拟页面没有实际交换进入物理存储器，直到CPU第一次引用到页面（也就是，发射一个虚拟地址，落在地址空间这个页面的范围之内）。如果区域比文件的这部分要大一些，那么就用零来填充这个区域的余下部分。

2. *匿名文件*：一个区域也可以映射到一个匿名文件，匿名文件是由内核创建的，包含的全是二进制零。CPU第一次引用这样一个区域内的虚拟页面时，内核就在物理存储器中找到一个合适的牺牲页面，如果该页面被修改过，就将这个页面换出来，用二进制零覆盖牺牲页面，并更新页表，将这个页面标记为是驻留在存储器中的。注意在磁盘和存储器之间并没有实际的数据传送。因为这个原因，映射到匿名文件的区域中的页面，有时也叫做*请求二进制零的页*（demand-zero page）。

无论在哪种情况中，一旦一个虚拟页面被初始化了，它就在一个由内核维护的专门的*交换文件*（swap file）之间换来换去。交换文件也叫做*交换空间*（swap space）或者*交换区域*（swap area）。需要意识到的很重要的一点是，在任何时刻，交换空间都限制着当前运行着的进程能够分配的虚拟页面的总数。

10.8.1 再看共享对象

存储器映射的概念来源于一个聪明的发现：如果虚拟存储器系统可以集成到传统的文件系统中，那么就能提供一种简单而高效的把程序和数据加载到存储器中的方法。

正如我们已经看到的，进程这一抽象能够为每个进程提供自己私有的虚拟地址空间，可以免受其他进程的错误读写。不过，许多进程有同样的只读文本区域。例如，每个运行 Unix shell 程序 tcsh 的进程都有相同的文本区域。而且，许多程序需要访问只读运行时库代码的相同拷贝。例如，每个 C 程序都要求来自标准 C 库的诸如 printf 这样的函数。那么，如果每个进程都在物理存储器中保持这些常用代码的复制拷贝，那就是极端的浪费了。幸运的是，存储器映射给我们提供了一种清晰的机制，用来控制多个进程如何共享对象。

一个对象可以被映射到虚拟存储器的一个区域，要么作为*共享对象*，要么作为*私有对象*。如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到它们虚拟存储器的其他进程而言，也是可见的。而且，这些变化也会反映在磁盘上的原始对象中。

另一方面，对于一个映射到私有对象的区域做的改变，对于其他进程来说是不可见的，并且进程对这个区域所做的任何写操作都不会反映在磁盘上的对象中。一个共享对象映射到的虚拟存储器区域叫做*共享区域*。类似地，也有*私有区域*。

假设进程 1 将一个共享对象映射到它的虚拟存储器的一个区域中，如图 10.31（a）所示。现在假设进程 2 将同一个共享对象映射到它的地址空间［并不一定要和进程 1 在相同的虚拟地址处，如图 10.31（b）所示］。

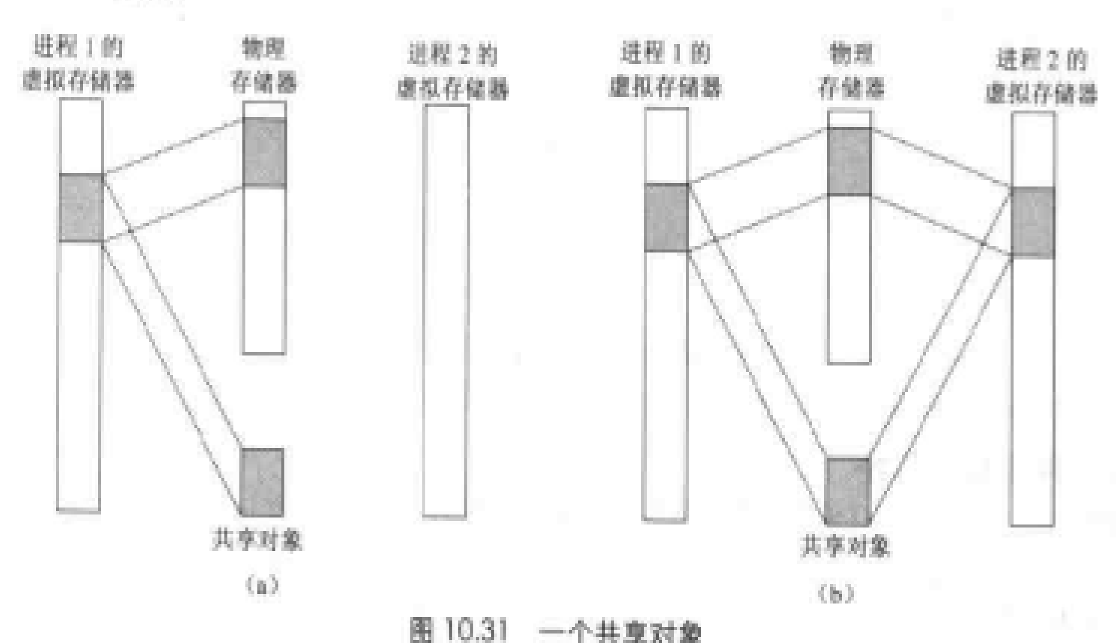


图 10.31 一个共享对象

（a）进程 1 映射了共享对象之后；（b）进程 2 映射了同一个共享对象之后。（注意物理页面不一定要求是连续的。）

因为每个对象都有一个惟一的文件名，内核可以迅速地判定进程 1 已经映射了这个对象，而且可以使进程 2 中的页表条目指向相应的物理页面。关键点在于即使对象被映射到了多个共享区域，物理存储器中也只需要存放共享对象的一个拷贝。为了方便，我们将物理页面显示为连续的，但是在一般情况下当然不是这样的。

私有对象是使用一种叫做写时拷贝（copy-on-write）的巧妙技术被映射到虚拟存储器中的。一个私有对象开始生命周期的方式基本上与共享对象的一样，在物理存储器中只保存有私有对象的一份拷贝。比如，图 10.32（a）展示了一种情况，其中两个进程将一个私有对象映射到它们虚拟存储器的不同区域，但是共享这个对象同一个物理拷贝。对于每个映射私有对象的进程，相应私有区域的页表条目都被标记为只读，并且区域结构被标记为私有的写时拷贝。只要没有进程试图写它自己的私有区域，它们就可以继续共享物理存储器中对象的一个单独拷贝。然而，只要有一个进程试图写私有区域内的某个页面，那么这个写操作就会触发一个保护故障。

当故障处理程序注意到保护异常是由于进程试图写私有的写时拷贝区域中的一个页面而引起的，它就会在物理存储器中创建这个页面的一个新拷贝，更新页表条目指向这个新的拷贝，然后恢复这个页面的可写权限，如图 10.32（b）所示。当故障处理程序返回时，CPU 重新执行这个写操作，现在在新创建的页面上这个写操作就可以正常执行了。

通过延迟私有对象中的拷贝直到最后可能的时刻，写时拷贝最充分地使用了稀有的物理存储器。

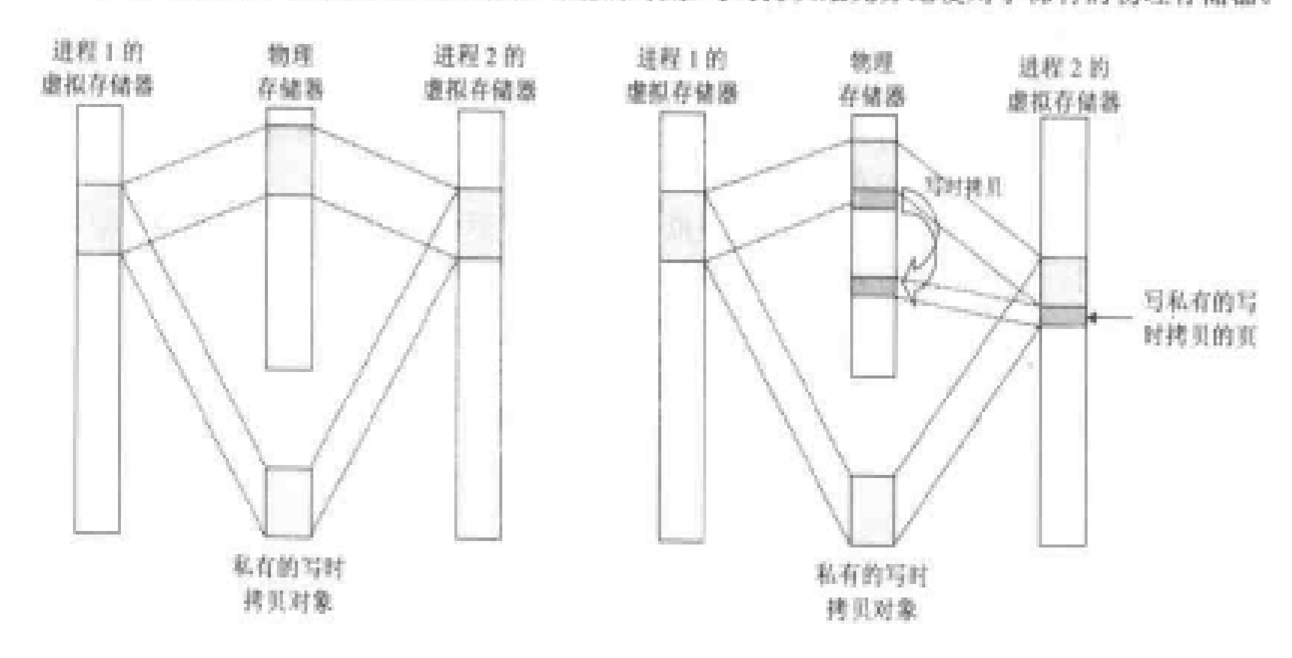


图 10.32　一个私有的写时拷贝对象

（a）两个进程都映射了私有的写时拷贝对象之后；（b）进程 2 写了私有区域中的一个页之后。

10.8.2　再看 fork 函数

既然我们理解了虚拟存储器和存储器映射，那么我们可以清晰地知道 fork 函数是如何创建一个带有自己独立虚拟地址空间的新进程的。

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个惟一的 PID。为了给这个新进程创建虚拟存储器，它创建了当前进程的 mm_struct、区域结构（vm_area_struct）和页表的原样拷贝。它标记两个进程中的每个页面为只读的，并标记两个进程中的每个区域结构为私有的写时拷贝的。

当 fork 在新进程中返回时，新进程现在的虚拟存储器刚好和调用 fork 时存在的虚拟存储器相同。当这两个进程中的任一个后来进行写操作时，写时拷贝机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

10.8.3 再看 execve 函数

虚拟存储器和存储器映射在将程序加载到存储器的过程中也扮演着关键的角色。既然我们已经理解了这些概念，我们就能够理解 execve 函数实际中是如何加载和执行程序的。假设运行在当前进程中的程序执行了如下的 execve 调用：

```
Execve("a.out", argv, environ);
```

execve 函数在当前进程中加载并运行包含在可执行目标文件 a.out 中的程序，用 a.out 程序有效地替代了当前程序。加载并运行 a.out 需要以下几个步骤：

- *删除已存在的用户区域*。删除当前进程虚拟地址的用户部分中的已存在的区域结构。
- *映射私有区域*。为新程序的文本、数据、bss 和栈区域创建新的区域结构。所有这些新的区域都是私有的写时拷贝的。文本和数据区域被映射为 a.out 文件中的文本和数据区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 a.out 中。栈和堆区域也是请求二进制零的，初始长度为零。图 10.33 概括了私有区域的不同映射。
- *映射共享区域*。如果 a.out 程序与共享对象（或目标）链接，比如标准 C 库 libc.so，那么这些对象都是动态链接到这个程序的，并且映射到用户虚拟地址空间中的共享区域内。
- *设置程序计数器(PC)*。execve 做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向文本区域的入口点。

下一次调度这个进程时，它将从这个入口点开始执行。Linux 将根据需要换入代码和数据页面。

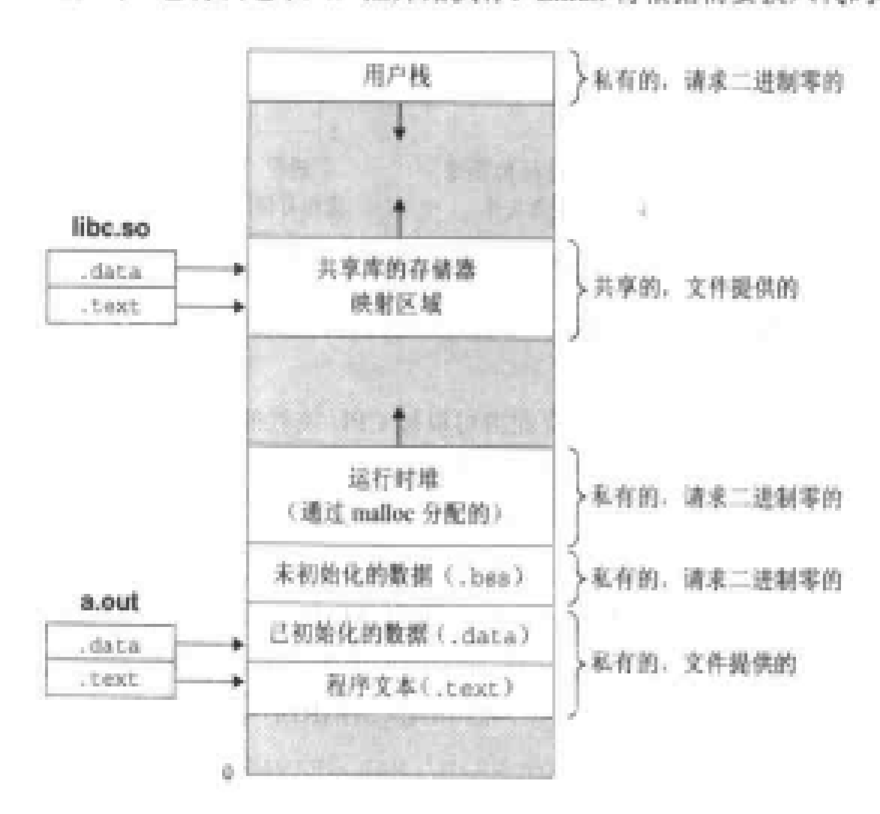


图 10.33 加载器是如何映射用户地址空间的区域的

10.8.4 使用 mmap 函数的用户级存储器映射

Unix 进程可以使用 mmap 函数来创建新的虚拟存储器区域，并将对象映射到这些区域中。

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot,
int flags, int fd, off_t offset);
```

返回：若成功时则为指向映射区域的指针，若出错则为-1.

mmap 函数要求内核创建一个新的虚拟存储器区域，最好是从地址 start 开始的一个区域，并将文件描述符 fd 指定的对象的一个连续的组块(chunk)映射到这个新的区域。连续的对象组块(chunk)大小为 length 字节，从距文件开始处偏移量为 offset 字节的地方开始。start 地址仅仅是一个暗示，通常被定义为 NULL。为了我们的目的，我们总是假设起始地址为 NULL。图 10.34 描述了这些参数的意义。

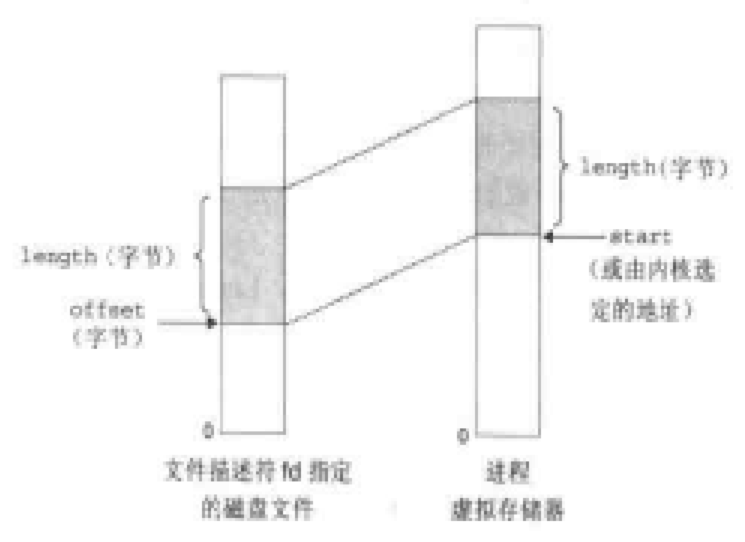


图 10.34 mmap 参数的可视化解释

参数 port 包含描述新映射的虚拟存储器区域的访问权限位（也就是，在相应区域结构中的 vm_port 位）。

- PROT_EXEC：这个区域内的页面由可以被 CPU 执行的指令组成。
- PROT_READ：这个区域内的页面可读。
- PROT_WRITE：这个区域内的页面可写。
- PROT_NONE：这个区域内的页面不能被访问。

参数 flags 由描述被映射对象类型的位组成。如果 MAP_ANON 标记位被设置，并且 fd 为 NULL，那么被映射的对象就是一个匿名对象，而相应的虚拟页面是请求二进制零的。MAP_PRIVATE 表示被映射的对象是一个私有的写时拷贝对象，而 MAP_SHARED 表示是一个共享对象。例如

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

让内核创建一个新的包含 size 字节的只读、私有、请求二进制零的虚拟存储器区域。如果调用成功，那么 bufp 包含新区域的地址。

munmap 函数删除虚拟存储器的区域：

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

返回：若成功则为 0，若出错则为 -1。

munmap 函数删除从虚拟地址 start 开始的，由接下来 length 字节组成的区域。接下来对已删除区域的引用会导致段错误。

练习题 10.5

编写一个 C 程序 mmapcopy.c，使用 mmap 将一个任意大小的磁盘文件拷贝到 stdout。输入文件的名字必须作为一个命令行参数来传递。

10.9 动态存储器分配

虽然可以使用低级的 mmap 和 munmap 函数来创建和删除虚拟存储器的区域，但是大多数 C 程序还是会在运行时需要额外虚拟存储器时，使用一种*动态存储器分配器*（dynamic memory allocator）。

一个动态存储器分配器维护着一个进程的虚拟存储器区域，称为*堆*（heap）（图 10.35）。在大多数的 Unix 系统中，堆是一个请求二进制零的区域，它紧接在未初始化的 bss 区域后开始，并向上生长（向更高的地址）。对于每个进程，内核维护着一个变量 brk（读做 "break"），它指向堆的顶部。

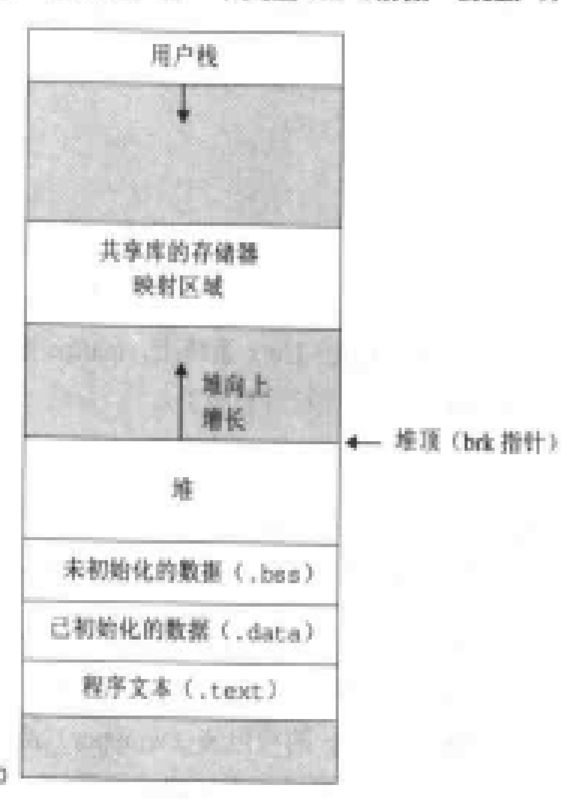


图 10.35 堆

分配器将堆视为一组不同大小的块（block）的集合来维护。每个块就是一个连续的虚拟存储器组块（chunk），要么是已分配的，要么是空闲的。已分配块（block）显式地保留为供应用使用。空

闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用显式执行的，要么是存储器分配器自身隐式执行的。

分配器有两种基本风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

显式分配器（explicit allocator）要求应用显式地释放任何已分配的块。例如，C 标准库提供一种叫做 malloc 程序包的显式分配器。C 程序通过调用 malloc 函数来分配一个块，并通过调用 free 函数来释放一个块。C++中的 new 和 delete 操作符与 C 中的 malloc 和 free 相当。

隐式分配器（implicit allocator），在另一方面，要求分配器检测何时一个已分配块不再被程序使用，然后就释放这个块。隐式分配器也叫做*垃圾收集器*（garbage collector），而自动释放未使用的已分配的块的过程叫做*垃圾收集*（garbage collection）。例如，诸如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

本节剩下的部分讨论的是显式分配器的设计和实现。我们将在 10.10 小节中讨论隐式分配器。为了更具体，我们的讨论集中于管理堆存储器的分配器。然而，学生们应该明白存储器分配是一个普遍的概念，可以出现在各种上下文中。例如，图形处理密集的应用程序就经常使用标准分配器来要求获得一大块虚拟存储器，然后使用与应用相关的分配器来管理块中的存储器，以支持图形节点的创建和销毁。

10.9.1 malloc 和 free 函数

C 标准库提供了一个称为 malloc 程序包的显式分配器。程序通过调用 malloc 函数来从堆中分配块。

```
#include <stdlib.h>

void *malloc(size_t size);
```

返回：若成功则为指针，若出错则为 NULL。

malloc 函数返回一个指针，指向大小为至少 size 字节的存储器块，这个块会为可能包含在这个块内的所有数据对象类型做对齐。在我们熟悉的 Unix 系统上，malloc 返回一个 8 字节（双字）边界对齐的块。size_t 类型被定义为 unsigned int（无符号整数）。

旁注：一个字有多大？

回想一下在第 3 章中我们对 IA32 机器代码的讨论，Intel 将 4 字节对象称为双字。然而，在本节中，我们会假设字是 4 字节的对象，而双字是 8 字节的对象，这和传统术语是一致的。

如果 malloc 遇到问题（例如，程序要求的存储器块比可用的虚拟存储器还要大），那么它就返回 NULL，并设置 errno。malloc 不初始化它返回的存储器。那些想要已初始化的动态存储器的应用程序可以使用 calloc，calloc 是一个基于 malloc 的瘦包装（wrapper）函数，它将分配的存储器初始化为零。想要改变一个以前已分配块的大小，可以使用 realloc 函数。

动态存储器分配器（例如 malloc）可以通过使用 mmap 和 munmap 函数，显式地分配和释放堆存储器，还可以使用 sbrk 函数：

```
#include <unistd.h>

void *sbrk(int incr);
```

返回：若成功则为老 brk 指针，若出错则为-1。

sbrk 函数通过将内核的 brk 指针增加 incr 来扩展和收缩堆。如果成功，它就返回 brk 的旧值，否则，它就返回-1，并将 errno 设置为 ENOMEM。如果 incr 为零，那么 sbrk 就返回 brk 的当前值。用一个为负的 incr 来调用 sbrk 是合法的，而且很巧妙，因为返回值（brk 的旧值）指向距新堆顶上面的 abs（incr）字节。

程序是通过调用 free 函数来释放已分配的堆块。

```
#include <stdlib.h>

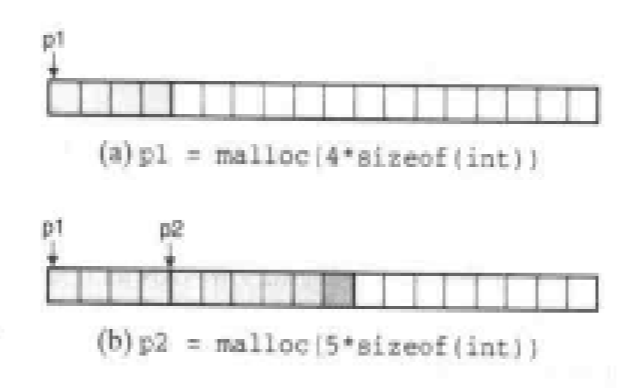
void free(void *ptr);
```

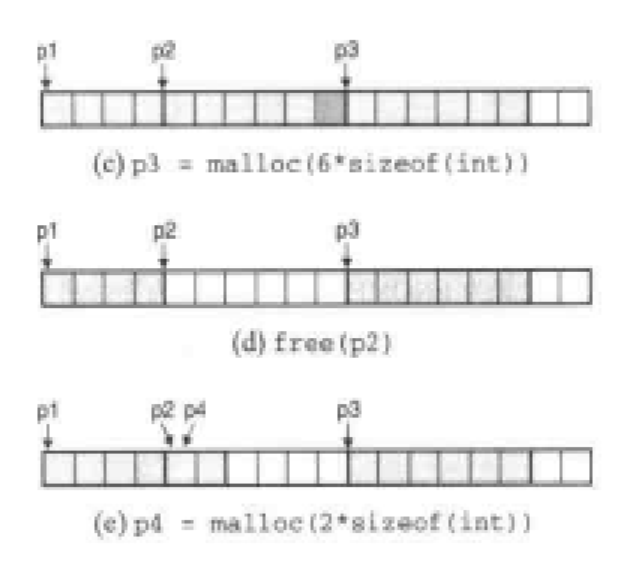
返回：无。

ptr 参数必须指向一个从 malloc 获得的已分配块的起始位置。如果不是，那么 free 的行为就是未定义的。更糟的是，既然它什么都不返回，free 就不会告诉应用出现了错误。就像我们将在 10.11 节里看到的，这会产生一些令人迷惑的运行时错误。

图 10.36 展示了一个 malloc 和 free 的实现是如何管理一个 C 程序的 16 字的（非常）小的堆的。每个方框代表了一个 4 字节的字。粗线标出的矩形对应于已分配块（有阴影的）和空闲块（无阴影的）。初始时，堆是由一个大小为 16 个字的、双字对齐的、空闲块组成的。

- 图 10.36（a）：程序请求一个 4 字的块。malloc 的响应是：从空闲块的前部切出一个 4 字的块，并返回一个指向这个块的第一字的指针。
- 图 10.36（b）：程序请求一个 5 字的块。malloc 的响应是：从空闲块的前部分配一个 6 字的块。在本例中，malloc 在块里填充了一个额外的字，是为了保持空闲块是双字边界对齐的。
- 图 10.36（c）：程序请求一个 6 字的块，而 malloc 就从空闲块的前部切出一个 6 字的块。
- 图 10.36（d）：程序释放在图 10.36（b）中分配的那个 6 字的块。注意，在调用 free 返回之后，指针 p2 仍然指向被释放了的块。应用有责任在它被一个新的 malloc 调用重新初始化之前，不再使用 p2。
- 图 10.36（e）：程序请求一个 2 字的块。在这种情况中，malloc 分配在前一步中被释放了的块的一部分，并返回一个指向这个新块的指针。





(c) p3 = malloc(6*sizeof(int))

(d) free(p2)

(e) p4 = malloc(2*sizeof(int))

图 10.36　用 malloc 分配和释放块

每个方框对应于一个字。每个粗线标出的矩形对应于一个块。已分配的块是有阴影的。空闲块是无阴影的。堆地址是从左往右增加的。

10.9.2　为什么要使用动态存储器分配？

程序使用动态存储器分配的最重要的原因是它们经常直到程序实际运行时，才知道某些数据结构的大小。例如，假设要求我们编写一个 C 程序，它读一个 *n* 个 ASCII 码整数的链表，每一行一个整数，从 stdin 到一个 C 数组。输入是由整数 *n*，和接下来要读和存储到数组中的 *n* 个整数组成的。最简单的方法就是用某种硬编码的最大数组大小静态地定义这个数组：

```
1   #include "csapp.h"
2   #define MAXN 15213
3
4   int array[MAXN];
5
6   int main()
7   {
8       int i, n;
9
10      scanf("%d", &n);
11      if (n > MAXN)
12          app_error("Input file too big");
13      for (i = 0; i < n; i++)
14          scanf("%d", &array[i]);
15      exit(0);
16  }
```

用这样硬编码的大小来分配数组通常不是种好想法。MAXN 的值是任意的，和机器上可用的虚拟存储器的实际数量没有关系。而且，如果这个程序的使用者想读取一个比 MAXN 大的文件，惟一的办法就是用一个更大的 MAXN 值来重新编译这个程序。虽然对于这个简单的示例来说这不成

问题，但是硬编码数组界限的出现对于拥有百万行代码和大量使用者的大型软件产品而言，会变成一场维护的噩梦。

一种更好的方法是在运行时，在已知了 n 的值之后，动态地分配这个数组。使用这种方法，数组大小的最大值就只由可用的虚拟存储器数量来限制了。

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int *array, i, n;
6
7       scanf("%d", &n);
8       array = (int *)Malloc(n * sizeof(int));
9       for (i = 0; i < n; i++)
10          scanf("%d", &array[i]);
11      exit(0);
12  }
```

动态存储器分配是一种有用而重要的编程技术。然而，为了正确而高效地使用分配器，程序员需要对它们是如何工作的有所了解。我们将在 10.11 节中讨论因为不正确地使用分配器所导致的一些可怕的错误。

10.9.3 分配器的要求和目标

显式分配器必须在一些相当严格的约束条件下工作：

- 处理任意请求序列。一个应用可以有任意序列的分配请求和释放请求，只要满足约束条件：每个释放请求必须对应于一个当前已分配块，这个块产生于以前的分配请求。因此，分配器不可以假设分配和释放请求的顺序。例如，分配器不能假设所有的分配请求都有相匹配的释放请求，或者有相匹配的分配和空闲请求是嵌套的。
- 立即响应请求。分配器必须立即响应分配请求。因此，不允许分配器为了提高性能重新排列或者缓冲请求。
- 只使用堆。为了使分配器是可扩展的，分配器使用的任何非标量数据结构都必须保存在堆里。
- 对齐块（对齐要求）。分配器必须对齐块，使得它们可以保存任何类型的数据对象。在大多数系统中，这意味着分配器返回的块是 8 字节（双字）边界对齐的。
- 不修改已分配的块。分配器只能操作或者改变空闲块。特别是，一旦块被分配了，就不允许修改或者移动它了。因此，诸如压缩已分配块这样的技术是不允许使用的。

在这些限制条件下工作，分配器的编写者试图实现吞吐率最大化和存储器使用率最大化，而这两个性能目标经常是相互冲突的。

- 目标 1：最大化吞吐率。假定 n 个分配和释放请求的某种序列：

$$R_0, R_1, \cdots, R_k, \cdots, R_{n-1}$$

我们希望一个分配器的吞吐率最大化，吞吐率就是在每个单位时间里完成的请求数。例

如，如果一个分配器在 1 秒中内完成 500 个分配请求和 500 个释放请求，那么它的吞吐率就是每秒 1000 次操作。一般而言，我们可以通过使满足分配和释放请求的平均时间最小化来使吞吐率最大化。正如我们会看到的，开发一个具有合理性能的分配器并不困难，所谓合理性能是指一个分配请求的最糟运行时间与空闲块的数量成线性关系，而一个释放请求的运行时间是个常数。

- 目标 2：*最大化存储器利用率*。天真的程序员经常不正确地假设虚拟存储器是一个无限的资源。实际上，一个系统中被所有进程分配的虚拟存储器的全部数量是受磁盘上交换空间的数量限制的。好的程序员知道虚拟存储器是一个有限的空间，必须高效地使用。对于可能被要求分配和释放大块存储器的动态存储器分配器来说，尤其如此。

  有很多方式来描述一个分配器使用堆的效率如何。在我们的经验中，最有用的标准是*峰值利用率*（peak utilization）。像以前一样，我们给定 n 个分配和释放请求的某种顺序

$$R_0, R_1, \cdots, R_k, \cdots, R_{n-1}$$

如果一个应用程序请求一个 p 字节的块，那么得到的已分配块的*有效载荷*（payload）是 p 字节。在请求 R_k 完成之后，*聚集有效载荷*（aggregate payload），表示为 P_k，为当前已分配的块的有效载荷之和，而 H_k 表示堆的当前的（单调不降低的）大小。

那么，前 k 个请求的*峰值利用率*，表示为 U_k，可以通过下式得到：

$$U_k = \frac{\max_{i \le k} P_i}{H_k}$$

那么，分配器的目标就是在整个序列中使峰值利用率 U_{n-1} 最大化。正如我们将要看到的，在最大化吞吐率和最大化利用率之间是有平衡关系的。特别是，以堆利用率为代价，很容易编写出吞吐率最大化的分配器。分配器设计中一个有趣的挑战就是在两个目标之间找到一个适当的平衡。

旁注：放宽单调性假设

我们可以通过让 H_k 成为前 k 个请求的最高峰，从而使得在我们对 U_k 的定义中放宽单调不降低的假设，并且允许堆增长和降低。

10.9.4 碎片

造成堆利用率很低的主要原因是一种称为*碎片*（fragmentation）的现象，当虽然有未使用的存储器但不能用来满足分配请求时，就发生这种现象。有两种形式的碎片：*内部碎片*（internal fragmentation）和*外部碎片*（external fragmentation）。

*内部碎片*是在一个已分配块比有效载荷大时发生的。很多原因都可能造成这个问题。例如，一个分配器的实现可能对已分配块强加一个最小的大小值，而这个大小要比某个请求的有效载荷大。或者，就如我们在图 10.36（b）中看到的，分配器可能增加块大小以满足对齐约束条件。

内部碎片的量化是简单明了的。它就是已分配块和它们的有效载荷之差的和。因此，在任意时刻，内部碎片的数量只取决于以前请求的模式和分配器的实现方式。

*外部碎片*是当空闲存储器合计起来足够满足一个分配请求，但是没有一个单独的空闲块足够大可以来处理这个请求时发生的。例如，如果图 10.36（e）中的请求要求 6 个字，而不是 2 个字，那么如果不向内核请求额外的虚拟存储器就无法满足这个请求，即使在堆中仍然有 6 个空闲的字。问

题的产生是由于这6个字是分在两个空闲块中的。

外部碎片比内部碎片的量化要困难得多，因为它不仅取决于以前请求的模式和分配器的实现方式，还取决于将来请求的模式。例如，假设在 k 个请求之后，所有空闲块的大小都恰好是4个字。这个堆会有外部碎片吗？答案取决于将来请求的模式。如果将来所有的分配请求都要求比4个字小的块，那么就不会有外部碎片。另一方面，如果有一个或者多个请求要求比4个字大的块，那么这个堆就会有外部碎片。

因为外部碎片是难以量化和不可能预测的，所以分配器典型地采用启发式策略来试图维持少量的大空闲块，而不是维持大量的小空闲块。

10.9.5 实现问题

可以想像出的最简单的分配器会把堆组织成一个大的字节数组，还有一个指针 P，初始指向这个数组的第一个字节。为了分配 size 字节，malloc 将 P 的当前值保存在栈里，将 P 增加 size，并将 P 的旧值返回到调用函数。free 只是简单地返回到调用函数，而不做其他任何事情。

这个简单的分配器是设计中的一种极端情况。因为每个 malloc 和 free 只执行很少量的指令，吞吐率会极好。然而，因为分配器从不重复使用任何块，存储器利用率将极差。一个实际的分配器要在吞吐率和利用率之间把握好平衡，就必须考虑以下几个问题：

- 空闲块组织：我们如何记录空闲块？
- 放置：我们如何选择一个合适的空闲块来放置一个新分配的块？
- 分割：在我们将一个新分配的块放置到某个空闲块之后，我们如何处理这个空闲块中的剩余部分？
- 合并：我们如何处理一个刚刚被释放的块？

本节剩下的部分将详细讨论这些问题。因为像放置、分割以及合并这样的基本技术贯穿在许多不同的空闲块组织中，所以我们将在一种叫做隐式空闲链表的简单空闲块组织结构中来介绍它们。

10.9.6 隐式空闲链表

任何实际的分配器都需要一些数据结构，允许它来区别块边界，并区别已分配块和空闲块。大多数分配器将这些信息嵌在块本身当中。一个简单的方法如图 10.37 所示。

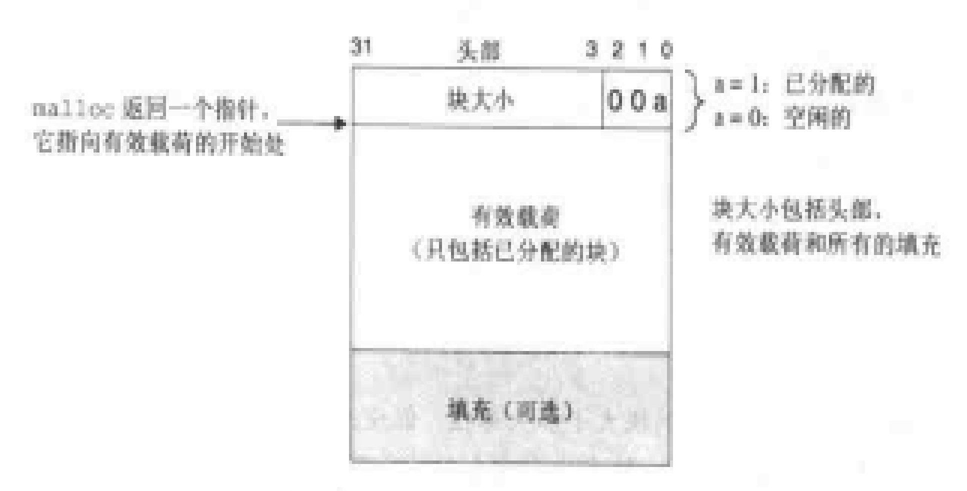


图 10.37 一个简单的堆块的格式

在这种情况中，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是8的倍数，且块大小的最低3位总是零。因此，我们只需要存储块大小的29个高位，释放剩余的3位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的，还是空闲的。例如，假设我们有一个已分配的块，大小为24（0x18）字节。那么它的头部将是

```
0x00000018 | 0x1 = 0x00000019.
```

类似地，一个块大小为40（0x28）字节的空闲块有如下的头部：

```
0x00000028 | 0x0 = 0x00000028.
```

头部后面就是应用调用malloc时请求的有效载荷。有效载荷后面是一块不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

假设块的格式如图10.37所示，我们可以将堆组织为一个连续的已分配块和空闲块的序列，如图10.38所示。

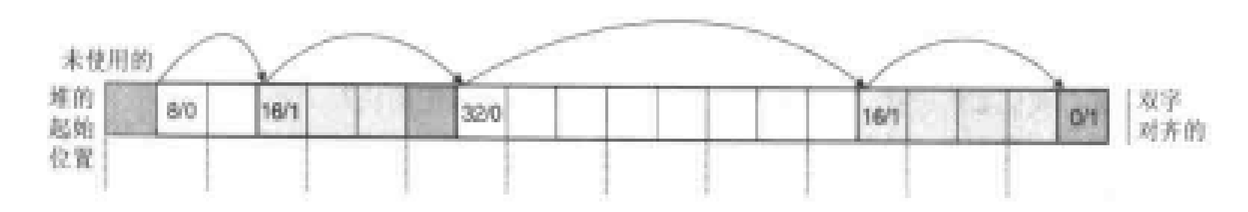


图10.38　用隐式空闲链表来组织堆

已分配块是有阴影的。空闲块是没有阴影的。头部标记为（大小（字节）/已分配位）。

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中的所有块，从而间接地遍历整个空闲块的集合。注意，我们需要以某种特殊标记结束的块，在这个示例中，就是一个设置了已分配位而大小为零的终止头部（terminating header）。（就像我们将在10.9.12节中看到的，设置已分配位简化了空闲块的合并。）

隐式空闲链表的优点是简单。显著的缺点是任何操作的开销，例如放置分配的块，要求空闲链表的搜索与堆中已分配块和空闲块的总数呈线性关系。

很重要的一点就是意识到系统对齐要求和分配器对块格式的选择对分配器上的最小块大小有强制的要求。没有已分配块或者空闲块可以比这个最小值还小。例如，如果我们假设一个双字的对齐要求，那么每个块的大小都必须是双字（8字节）的倍数。因此，图10.37中的块格式就导致最小的块大小为两个字：一个字作头，另一个字维持对齐要求。即使应用只请求一字节，分配器也仍然需要创建一个两字的块。

练习题 10.6

确定下面malloc请求序列产生的块大小和头部值。假设：分配器保持双字对齐，并且使用块格式如图10.37中所示的隐式空闲链表；块大小向上舍入为最接近的8字节的倍数。

请求	块大小（十进制字节）	块头部（十六进制）
malloc(1)		
malloc(5)		
malloc(12)		
malloc(13)		

10.9.7 放置分配的块

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大、可以放置所请求块的空闲块。分配器执行这种搜索的方式是由**放置策略**（placement policy）确定的。一些常见的策略是**首次适配**（first fit）、**下一次适配**（next fit）和**最佳适配**（best fit）。

首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配很相似，只不过不是从链表的起始处开始每次搜索，而是从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择匹配所需请求大小的最小空闲块。

首次适配的一个优点是它趋向于将大的空闲块保留在链表的后面。缺点是它趋向于在靠近链表起始处留下小空闲块的"碎片"，这就增加了对较大块的搜索时间。下一次适配是由 Donald Knuth 作为首次适配的一种代替品最早提出的，源于这样一个想法：如果我们上一次在某个空闲块里已经发现了一个匹配，那么很可能下一次我们也能在这个剩余块中发现匹配。下一次适配比首次适配运行起来明显要快一些。然而，一些研究表明，下一次适配的存储器利用率要比首次适配低得多。研究还表明最佳适配比首次适配和下一次适配的利用率都要高一些。然而，在简单空闲链表组织结构中，比如隐式空闲链表中，使用最佳适配的缺点是它要求对堆进行彻底的搜索。在后面，我们将看到更加精细复杂的分离式空闲链表组织，它实现了最佳适配策略，而不需要进行彻底的堆搜索。

10.9.8 分割空闲块

一旦分配器找到一个匹配的空闲块，它就必须做另一个策略决定，那就是分配这个空闲块中多少空间。一个选择是用整个空闲块。虽然这种方式简单而快捷，但是主要的缺点就是它会造成内部碎片。如果放置策略趋向于产生好的匹配，那么额外的内部碎片也是可以接受的。

然而，如果匹配不太好，那么分配器通常会选择将这个空闲块分割为两部分。第一部分变成分配块，而剩下的变成一个新的空闲块。图 10.39 展示了分配器如何分割图 10.38 中 8 个字的空闲块，来满足一个应用的对堆存储器 3 个字的请求。

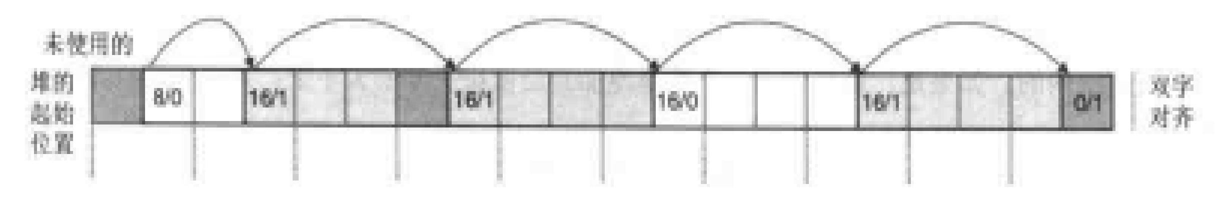


图 10.39 分割一个空闲块，以满足一个 3 个字的分配请求

已分配块是有阴影的。空闲块是没有阴影的。头部标记为（大小（字节）/已分配位）。

10.9.9 获取额外的堆存储器

如果分配器不能为请求块找到合适的空闲块，将发生什么呢？一个选择是通过合并那些在存储

器中物理上相邻的空闲块来创建一些更大的空闲块（在下一节中描述）。然而，如果这样还是不能生成一个足够大的块，或者如果空闲块已经最大程度地合并了，那么分配器就会向内核请求额外的堆存储器，要么是通过调用 mmap，要么是通过调用 sbrk 函数。在任一种情况下，分配器都会将额外的（或增加的）存储器转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

10.9.10 合并空闲块

当分配器释放一个已分配块时，可能有其他空闲块与这个新释放的空闲块相邻。这些邻接的空闲块可能引起一种现象，叫做假碎片（fault fragmentation），这里有许多可用的空闲块被切割成为小的、无法使用的空闲块。比如，图 10.40 展示了释放图 10.39 中分配的块后得到的结果。结果是两个相邻的空闲块，每个的有效载荷都为 3 个字。因此，接下来一个对 4 字有效载荷的请求就会失败，即使两个空闲块的合计大小足够大，可以满足这个请求。

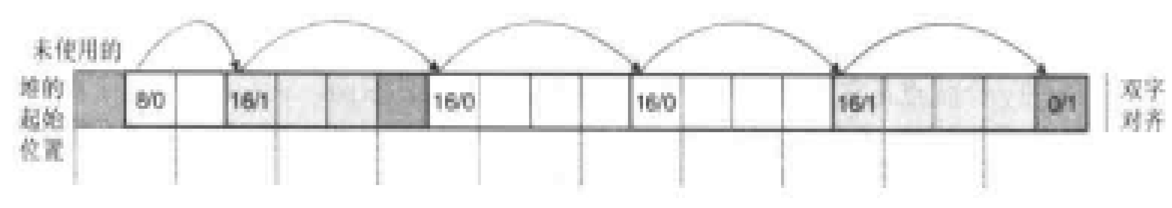


图 10.40 假碎片的示例

已分配块是有阴影的空闲块是没有阴影的。头部标记为（大小（字节）/已分配位）。

为了对付假碎片问题，任何实际的分配器都必须合并相邻的空闲块，这个过程称为合并（coalescing）。这就提出了一个重要的策略决定，那就是何时执行合并。分配器可以选择立即合并（immediate coalescing），也就是在每次一个块被释放时，就合并所有的相邻块。或者它也可以选择推迟合并（deferred coalescing），也就是等到某个稍晚的时候再合并空闲块。例如，分配器可以推迟合并，直到某个分配请求失败，然后扫描整个堆，合并所有的空闲块。

立即合并很简单明了，可以在常数时间内执行完成，但是对于某些请求模式，这种方式会产生一种形式的抖动，块会反复地合并，然后马上分割。例如，在图 10.40 中，反复地分配和释放一个 3 个字的块将产生大量不必要的分割和合并。在我们对分配器的讨论中，我们会假设使用立即合并，但是你应该了解，快速的分配器通常会选择某种形式的推迟合并。

10.9.11 带边界标记的合并

分配器是如何实现合并的？让我们称我们想要释放的块为当前块。那么，合并（存储器中的）下一个空闲块很简单而且高效。当前块的头部指向下一个块的头部，可以检查这个指针以判断下一个块是否是空闲的。如果是，就将它的大小简单地加到当前块头部的大小上，这两个块在常数时间内被合并。

但是我们该如何合并前面的块呢？给定一个带头部的隐式空闲链表，惟一的选择将是搜索整个链表，记住前面块的位置，直到我们到达当前块。使用隐式空闲链表，这意味着每次调用 free 的时间都与堆的大小成线性关系。即使使用更复杂精细的空闲链表组织，搜索时间也不会是常数。

Knuth 提出了一种聪明而通用的技术，叫做边界标记（boundary tag），允许在常数时间内进行对前面块的合并。这种思想，如图 10.41 所示，是在每个块的结尾处添加一个脚部（footer 边界标记），

其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块结尾位置[1]一个字的距离。



图 10.41 使用边界标记的堆块的格式

考虑当分配器释放当前块时所有可能存在的情况：

1. 前面的块和后面的块都是已分配的。
2. 前面的块是已分配的，后面的块是空闲的。
3. 前面的块是空闲的，而后面的块是已分配的。
4. 前面的和后面的块都是空闲的。

图 10.42 展示了我们如何对这四种情况进行合并。

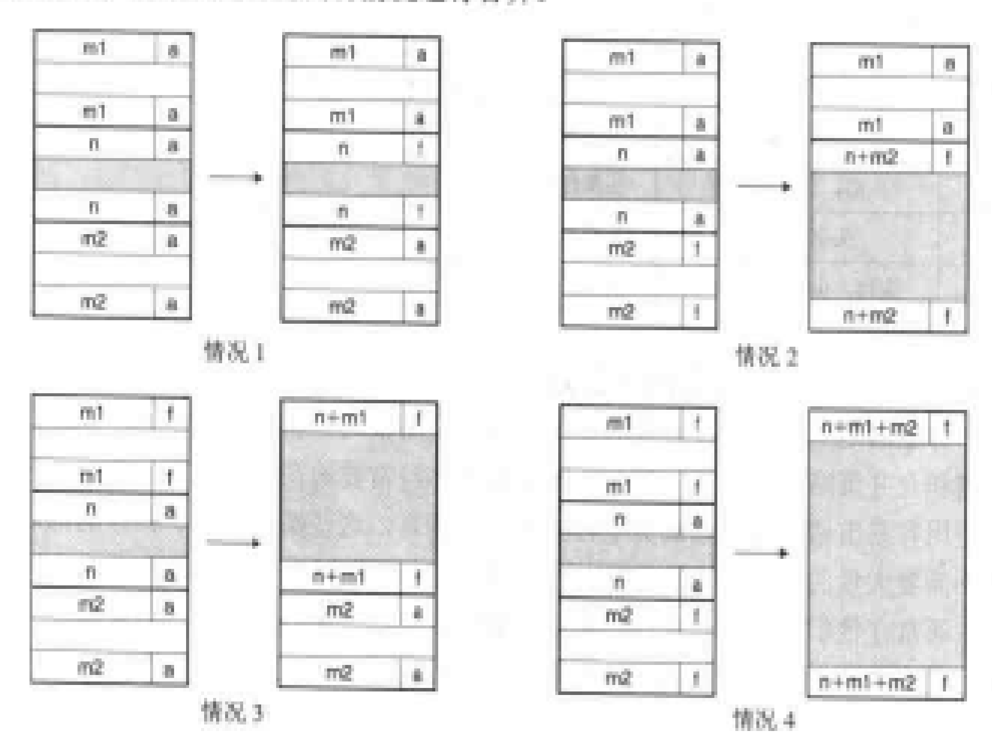


图 10.42 使用边界标记的合并

情况 1：前面的和后面块都已分配。情况 2：前面块已分配，后面块空闲。情况 3：前面块空闲，后面块已分配。情况 4：后面块和前面块都空闲。

1 原文为开始位置。——译者

在情况1中，两个邻接的块都是已分配的，因此不可能进行合并。所以当前块的状态仅仅是从已分配变成空闲。在情况2中，当前块与后面的块合并。用当前块和后面块的大小的和来更新当前块的头部和后面块的脚部。在情况3中，前面的块和当前块合并。用两个块大小的和来更新前面块的头部和当前块的脚部。在情况4中，要合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。在每种情况中，合并都是在常数时间内完成的。

边界标记的概念是简单优雅的，它对许多不同类型的分配器和空闲链表组织都是通用的。然而，它也存在一个潜在的缺陷。要求每个块都保持一个头部和一个脚部，在应用程序操作许多个小块时，会产生显著的存储器开销。例如，如果一个图形应用通过反复调用 malloc 和 free，来动态地创建和销毁图形节点，并且每个图形节点都只要求两个存储器字，那么头部和脚部将占用每个已分配块的一半的空间。

幸运的是，有一种非常聪明的边界标记的优化方法，能够使得在已分配块中不再需要脚部。回想一下，当我们试图在存储器中合并当前块以及前面的块和后面的块时，只有在前面的块是空闲时，才会需要用到它的脚部。如果我们把前面块的已分配/空闲位存放在当前块中多出来的低位中，那么已分配的块[2]就不需要脚部了，这样我们就可以将这个多出来的空间用作有效载荷了。不过请注意，空闲块[3]仍然需要脚部。

练习题 10.7

确定下面每种对齐要求和块格式的组合的最小的块大小。假设：隐式空闲链表，不允许有效载荷为零，头部和脚部存放在四字节的字中。

对齐要求	已分配的块	空闲块	最小块大小（字节）
单字	头部和脚部	头部和脚部	
单字	头部，但是无脚部	头部和脚部	
双字	头部和脚部	头部和脚部	
双字	头部，但是没有脚部	头部和脚部	

10.9.12 综合：实现一个简单的分配器

构造一个分配器是一件富有挑战性的任务。设计空间很大，有多种块格式、空闲链表格式，以及放置、分割和合并策略可供选择。另一个挑战就是你经常被迫在类型系统的安全和熟悉的限定之外编程，即使用容易出错的指针强制类型转换和指针运算，这些操作都属于典型的低层系统编程。虽然分配器不需要大量的代码，但是它们也还是细微而不可忽视的。熟悉诸如 C++或者 Java 之类高级语言的学生通常在他们第一次遇到这种类型的编程时，会遭遇一个概念上的障碍。为了帮助你清除这个障碍，我们将基于隐式空闲链表，使用立即边界标记合并方式，从头至尾地讲述一个简单分配器的实现。

一般分配器设计

我们的分配器使用如图 10.43 所示的 memlib.c 包所提供的一个存储器系统模型。模型的目的在

2 指前块。——译者
3 指前块。——译者

于允许我们在不干涉已存在的系统层 malloc 包的情况下，运行我们的分配器。

code/vm/memlib.c

```
1   #include "csapp.h"
2
3   /* private global variables */
4   static char *mem_start_brk;  /* points to first byte of the heap */
5   static char *mem_brk;        /* points to last byte of the heap */
6   static char *mem_max_addr;   /* max virtual address for the heap */
7
8   /*
9    * mem_init - initializes the memory system model
10   */
11  void mem_init(int size)
12  {
13      mem_start_brk = (char *)Malloc(size);       /* models available VM */
14      mem_brk = mem_start_brk;                    /* heap is initially empty */
15      mem_max_addr = mem_start_brk + size;        /* max VM address for heap */
16  }
17
18  /*
19   * mem_sbrk - simple model of the the sbrk function. Extends the heap
20   *      by incr bytes and returns the start address of the new area. In
21   *      this model, the heap cannot be shrunk.
22   */
23  void *mem_sbrk(int incr)
24  {
25      char *old_brk = mem_brk;
26
27      if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
28          errno = ENOMEM;
29          return (void *)-1;
30      }
31      mem_brk += incr;
32      return old_brk;
33  }
```

code/vm/memlib.c

图 10.43 memlib.c：存储器系统模型

mem_init 函数将堆可用的虚拟存储器模型化为一个大的、双字对齐的字节数组。在 mem_start_brk 和 mem_brk 之间的字节表示已分配的虚拟存储器。mem_brk 之后的字节表示未分配的虚拟存储器。分配器通过调用 mem_sbrk 函数来请求额外的堆存储器，这个函数和系统的 sbrk 函

数的接口相同，而且语义也相同，除了它会拒绝收缩堆的请求。

分配器包含在一个源文件中（malloc.c），用户可以编译和链接这个源文件到他们的应用之中。分配器输出三个函数到应用程序：

```
1   int mm_init(void);
2   void *mm_malloc(size_t size);
3   void mm_free(void *bp);
```

mm_init 函数初始化分配器，如果成功就返回 0，否则就返回-1。mm_malloc 和 mm_free 函数与它们对应的系统函数有相同的接口和语义。分配器使用如图 10.41 所示的块格式。最小块的大小为 16 字节。空闲链表组织成为一个隐式空闲链表，具有如图 10.44 所示的恒定形式（invariant form）。

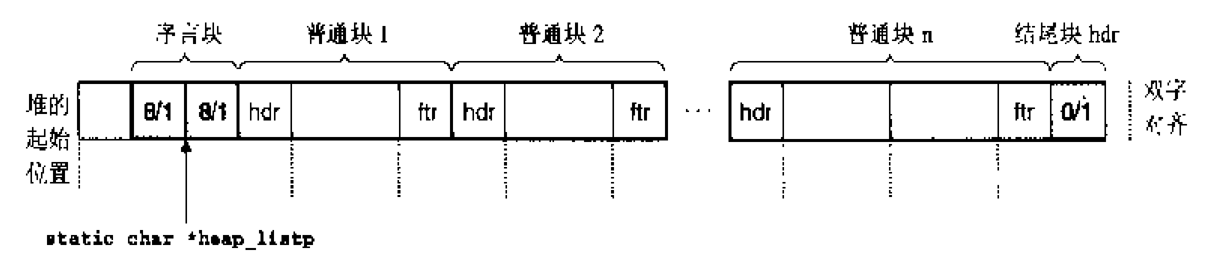


图 10.44 隐式空闲链表的恒定形式

第一个字是一个双字边界对齐的不使用的填充字。填充后面紧跟着一个特殊的序言块（prologue block），这是一个 8 字节的已分配块，只由一个头部和一个脚部组成。序言块是在初始化时创建的，并且永不释放。在序言块后紧跟的是零个或者多个由 malloc 或者 free 调用创建的普通块。堆总是以一个特殊的结尾块（epilogue block）来结束，这个块是一个大小为零的已分配块，只由一个头部组成。序言块和结尾块是一种消除合并时边界条件的技巧。分配器使用一个单独的私有（静态）全局变量（heap_listp），它总是指向序言块。（作为一个小优化，我们可以让它指向下一个块，而不是这个序言块。）

操作空闲链表的基本常数和宏

图 10.45 展示了一些我们在分配器编码中将要使用的基本常数。

code/vm/malloc.c

```
1   /* Basic constants and macros */
2   #define WSIZE       4       /* word size (bytes) */
3   #define DSIZE       8       /* doubleword size (bytes) */
4   #define CHUNKSIZE  (1<<12)  /* initial heap size (bytes) */
5   #define OVERHEAD    8       /* overhead of header and footer (bytes) */
6
7   #define MAX(x, y) ((x) > (y)? (x) : (y))
8
9   /* Pack a size and allocated bit into a word */
10  #define PACK(size, alloc)  ((size) | (alloc))
11
12  /* Read and write a word at address p */
```

```
13  #define GET(p)              (*(size_t *)(p))
14  #define PUT(p, val)         (*(size_t *)(p) = (val))
15
16  /* Read the size and allocated fields from address p */
17  #define GET_SIZE(p)     (GET(p) & ~0x7)
18  #define GET_ALLOC(p)    (GET(p) & 0x1)
19
20  /* Given block ptr bp, compute address of its header and footer */
21  #define HDRP(bp)      ((char *)(bp) - WSIZE)
22  #define FTRP(bp)      ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
23
24  /* Given block ptr bp, compute address of next and previous blocks */
25  #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
26  #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```
code/vm/malloc.c

图 10.45 操作空闲链表的基本常数和宏

第 2~5 行定义了一些基本的大小常数：字的大小（WSIZE）和双字的大小（DSIZE），初始空闲块的大小和扩展堆时的默认大小（CHUNKSIZE），以及头部和脚部占用的开销字节数量（OVERHEAD）。

在空闲链表中操作头部和脚部可能是很麻烦的，因为它要求大量使用强制类型转换和指针运算。因此，我们发现定义一个小的宏的集合来访问和遍历空闲链表是很有帮助的（第 10~26 行）。PACK 宏（第 10 行）将大小和已分配位结合起来，并返回一个值，可以把它存放在头部或者脚部中。

GET 宏（第 13 行）读取和返回参数 p 引用的字。这里强制类型转换是至关重要的。参数 p 典型地是一个（viod *）指针，不可以直接进行间接引用。类似地，PUT 宏（第 14 行）将 val 存放在参数 p 指向的字中。

GET_SIZE 和 GET_ALLOC 宏（第 17~18 行）从地址 p 处的头部或者脚部，分别返回大小和已分配位。剩下的宏是对块指针（block pointer，用 bp 表示）的操作，块指针指向第一个有效载荷字节。给定一个块指针 bp，HDRP 和 FTRP 宏（第 21~22 行）分别返回指向这个块的头部和脚部的指针。NEXT_BLKP 和 PREV_BLKP 宏（第 25~26 行）分别返回指向后面的块和前面的块的块指针。

可以以多种方式来编辑宏，以操作空闲链表。比如，给定一个指向当前块的指针 bp，我们可以使用下面的代码行来确定存储器中后面的块的大小：

```
size_t size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
```

创建初始空闲链表

在调用 mm_malloc 或者 mm_free 之前，应用必须通过调用 mm_init 函数来初始化堆（参见图 10.46）。

code/vm/malloc.c
```
1   int mm_init(void)
```

```
 2   {
 3       /* create the initial empty heap */
 4       if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
 5           return -1;
 6       PUT(heap_listp, 0);                          /* alignment padding */
 7       PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));    /* prologue header */
 8       PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));    /* prologue footer */
 9       PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));     /* epilogue header */
10       heap_listp += DSIZE;
11
12       /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13       if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14           return -1;
15       return 0;
16   }
```

code/vm/malloc.c

图 10.46 mm_init：创建一个带初始空闲块的堆

mm_init 函数从存储器系统得到 4 个字，并将它们初始化，从而创建一个空的空闲链表（第 4～10 行）。然后它调用 extend_heap 函数（图 10.47），这个函数将堆扩展 CHUNKSIZE 字节，并且创建初始的空闲块。此刻，分配器已初始化了，并且准备好接受来自应用的分配和释放请求。

code/vm/malloc.c

```
 1   static void *extend_heap(size_t words)
 2   {
 3       char *bp;
 4       size_t size;
 5
 6       /* Allocate an even number of words to maintain alignment */
 7       size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
 8       if ((int)(bp = mem_sbrk(size)) < 0)
 9           return NULL;
10
11       /* Initialize free block header/footer and the epilogue header */
12       PUT(HDRP(bp), PACK(size, 0));                /* free block header */
13       PUT(FTRP(bp), PACK(size, 0));                /* free block footer */
14       PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));        /* new epilogue header */
15
16       /* Coalesce if the previous block was free */
17       return coalesce(bp);
18   }
```

code/vm/malloc.c

图 10.47 extend_heap：用一个新的空闲块扩展堆

extend_heap 函数会在两种不同的环境中被调用：①当堆被初始化时；②当 mm_malloc 不能找

到一个合适的匹配块时。为了保持对齐，extend_heap 将请求大小向上舍入为最接近的 2 字（8 字节）的倍数，然后向存储器系统请求额外的堆空间（第 7~9 行）。

extend_heap 函数的剩余部分（第 12~17 行）在某些方面是很细微的。堆开始于一个双字对齐的边界，并且每次对 extend_heap 的调用都返回一个块，该块的大小是双字的整数倍。因此，对 mem_sbrk 的每次调用都返回一个双字对齐的存储器组块（chunk），紧跟在结尾块的头部后面。这个头部变成了新的空闲块的头部（第 12 行），并且这个组块（chunk）的最后一个字变成了新的结尾块的头部（第 14 行）。最后，在很可能出现的前一个堆以一个空闲块结束的情况中，我们调用 coalesce 函数来合并两个空闲块，并返回指向合并后的块的块指针（第 17 行）。

释放和合并块

应用通过调用 mm_free 函数（图 10.48），来释放一个以前分配的块，这个函数释放所请求的块（bp），然后使用 10.9.11 节中描述的边界标记合并技术将之与邻接的空闲块合并起来。

code/vm/malloc.c

```
1   void mm_free(void *bp)
2   {
3       size_t size = GET_SIZE(HDRP(bp));
4
5       PUT(HDRP(bp), PACK(size, 0));
6       PUT(FTRP(bp), PACK(size, 0));
7       coalesce(bp);
8   }
9
10  static void *coalesce(void *bp)
11  {
12      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14      size_t size = GET_SIZE(HDRP(bp));
15
16      if (prev_alloc && next_alloc) {              /* Case 1 */
17          return bp;
18      }
19
20      else if (prev_alloc && !next_alloc) {        /* Case 2 */
21          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22          PUT(HDRP(bp), PACK(size, 0));
23          PUT(FTRP(bp), PACK(size,0));
24          return(bp);
25      }
26
27      else if (!prev_alloc && next_alloc) {        /* Case 3 */
28          size += GET_SIZE(HDRP(PREV_BLKP(bp)));
29          PUT(FTRP(bp), PACK(size, 0));
30          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
```

```
31          return(PREV_BLKP(bp));
32      }
33
34      else {                                          /* Case 4 */
35        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
36            GET_SIZE(FTRP(NEXT_BLKP(bp)));
37        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
38        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
39        return(PREV_BLKP(bp));
40      }
41  }
```
code/vm/malloc.c

图 10.48 mm_free：释放一个块，并使用边界标记合并将之与所有的邻接空闲块在常数时间内合并起来

coalesce 函数中的代码是图 10.42 中勾画的四种情况的一种简单直接的实现方式。这里也有些细微的方面。我们选择的空闲链表格式——它的序言块和结尾块总是标记为已分配——允许我们忽略潜在的麻烦边界情况，也就是，请求块 bp 在堆的起始处或者是在堆的结尾处。如果没有这些特殊块，代码将混乱得多，更加容易出错，并且更慢，因为我们将不得不在每次释放请求时，都去检查这些并不常见的边界情况。

分配块

一个应用通过调用 mm_malloc 函数（图 10.49）来向存储器请求大小为 size 字节的块。在检查完请求的真假之后（第 8～9 行），分配器必须调整请求块的大小，从而为头部和脚部留有空间，并满足双字对齐要求。第 12～13 行强制了最小块大小是 16 字节：8 字节（DSIZE）用来满足对齐要求，而另外 8 个（OVERHEAD）用来放头部和脚部。对于超过 8 字节的请求（第 15 行），一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍（DSIZE）。

code/vm/malloc.c

```
1   void *mm_malloc(size_t size)
2   {
3       size_t asize;        /* adjusted block size */
4       size_t extendsize;   /* amount to extend heap if no fit */
5       char *bp;
6
7       /* Ignore spurious requests */
8       if (size <= 0)
9           return NULL;
10
11      /* Adjust block size to include overhead and alignment reqs. */
12      if (size <= DSIZE)
13          asize = DSIZE + OVERHEAD;
14      else
15          asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);
16
17      /* Search the free list for a fit */
```

```
18        if ((bp = find_fit(asize)) != NULL) {
19            place(bp, asize);
20            return bp;
21        }
22
23        /* No fit found. Get more memory and place the block */
24        extendsize = MAX(asize,CHUNKSIZE);
25        if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26            return NULL;
27        place(bp, asize);
28        return bp;
29    }
```

code/vm/malloc.c

图 10.49 mm_malloc：从空闲链表分配一个块

一旦分配器调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块（第 18 行）。如果有合适的，那么分配器就放置这个请求块，并有选择地分割出多余部分（第 19 行），然后返回新分配块的地址（第 20 行）。

如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆（第 24～26 行），把请求块放置在这个新的空闲块里，有选择地分割这个块（第 27 行），然后返回一个指针，指向这个新分配的块（第 28 行）。

练习题 10.8

为 10.9.12 节中描述的简单分配器实现一个 find_fit 函数。

```
static void *find_fit(size_t asize)
```

你的解答应该对隐式空闲链表执行首次适配搜索。

练习题 10.9

为示例的分配器编写一个 place 函数。

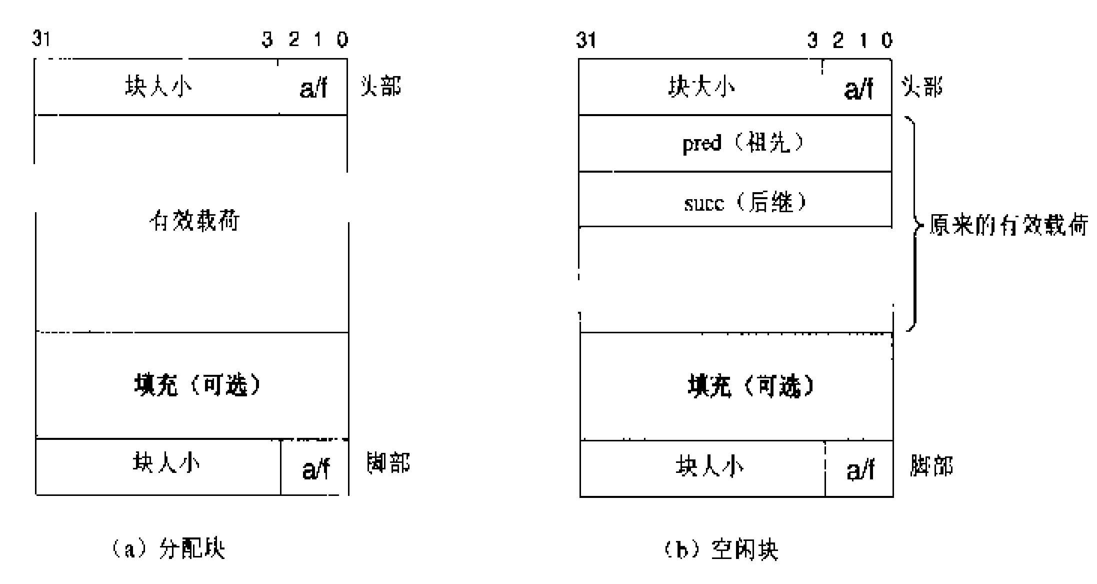
```
static void place(void *bp, size_t asize)
```

你的解答应该将请求块放置在空闲块的起始位置，只有当剩余部分的大小等于或者超出最小块的大小时，才进行分割。

10.9.13 显式空闲链表

隐式空闲链表为我们提供了一种简单的介绍一些基本分配器概念的方法。然而，因为块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表是不适合的（尽管对于堆块数量预先就知道是很小的特殊的分配器来说，它是比较好的）。

一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序是不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（祖先）和 succ（后继）指针，如图 10.50 所示。



（a）分配块　　（b）空闲块

图 10.50 使用双向空闲链表的堆块的格式

使用双向链表，而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于我们在空闲链表中对块排序所选择的策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它祖先的地址。在这种情况下，释放一个块需要线性时间的搜索，来定位合适的祖先。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的存储器利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

10.9.14 分离的空闲链表

就像我们已经看到的，一个使用单向空闲块链表的分配器需要与空闲块数量成线性关系的时间来分配块。一种流行的减少分配时间的方法，通常称为*分离存储*（segregated storage），维护多个空闲链表，其中每个链表中的块有大致相等的大小。

一般的思路是将所有可能的块大小分成一些等价类，也叫做*大小类*（size class）。有很多种方式来定义大小类。例如，我们可以根据 2 的幂来划分块大小：

$$\{1\},\{2\},\{3,4\},\{5\text{-}8\},\cdots,\{1025\text{-}2048\},\{2049\text{-}4096\},\{4097\text{-}\infty\}$$

或者我们可以将小的块分派到它们自己的大小类里，而将大块按照 2 的幂分类：

$$\{1\},\{2\},\{3\},\cdots,\{1023\},\{1024\},\cdots,\{1025\text{-}2048\},$$
$$\{2049\text{-}4096\},\{4097\text{-}\infty\}$$

分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为 n 的块时，它就搜索相应的空闲链表。如果它不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。

有关动态存储分配的文献描述了很多种分离存储方法，主要的区别在于它们如何定义大小类，何时进行合并，何时向操作系统请求额外的堆存储器，是否允许分割，等等。为了使你大致了解有哪些可能性，我们会描述两种基本的方法：简单分离存储（simple segregated storage）和分离适配（segregated fit）。

简单分离存储

使用简单分离存储，每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。例如，如果某个大小类定义为{17-32}，那么这个类的空闲链表全由大小为 32 的块组成。

为了分配一个给定大小的块，我们检查相应的空闲链表。如果链表非空，我们简单地分配其中第一块的全部。空闲块是不会分割以满足分配请求的。如果链表为空，分配器就向操作系统请求一个固定大小的额外存储器组块（典型地是页面大小的整数倍），将这个组块（chunk）分成大小相等的块，并将这些块链接起来形成新的空闲链表。要释放一个块，分配器只要简单地将这个块插入到相应的空闲链表的前部。

这种简单方法有许多优点。分配和释放块都是很快的常数时间操作。而且，每个组块（chunk）中都是大小相等的块，不分割，不合并，这意味着每个块只有很少的存储器开销。既然每个组块只有大小相同的块，那么一个已分配块的大小就可以从它的地址中推断出来。因为没有合并，所以已分配块的头部就不需要一个已分配/空闲标记。因此已分配块不需要头部，同时因为没有合并，它们也不需要脚部。因为分配和释放操作都是在空闲链表的起始处操作，所以链表只需要是单向的，而不用是双向的了。关键点在于，惟一在任何块中都需要的字段是每个空闲块中的一个字的 succ 指针，因此最小块大小就是一个字。

一个显著的缺点是，简单分离存储很容易造成内部和外部碎片。因为空闲块是不会被分割的，所以可能会造成内部碎片。更糟的是，某些引用模式会引起极多的外部碎片，因为是不会合并空闲块的（练习题 10.10）。

研究者提出了一种粗糙的合并形式来对付外部碎片问题。分配器记录操作系统返回的每个存储器组块（chunk）中的空闲块的数量。无论何时，如果有一个组块完全由空闲块组成，那么分配器就从它的当前大小类中删除这个组块，使得它对其他大小类可用。

练习题 10.10

描述一个在基于简单分离存储的分配器中会导致严重外部碎片的引用模式。

分离适配

使用这种方法，分配器维护着一个空闲链表的数组。每个空闲链表是和一个大小类相关联的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小是大小类的成员。有许多种不同的分离适配分配器。这里，我们描述了一种简单的版本。

为了分配一个块，我们必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果我们找到了一个，那么我们（可选地）分割它，并将剩余的部分插入到适当的空闲链表中。如果我们找不到合适的块，那么我们就搜索下一个更大的大小类的空闲链表。如此重复，直到找到一个合适的块。如果没有空闲链表中有合适的块，那么我们就向操作系统请求额外的堆存

储器，从这个新的堆存储器中分配出一个块，将剩余的部分放置在最大的大小类中。要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

分离适配方法是一种常见的选择，C 标准库中提供的 GNU malloc 包就是采用的这种方法，因为这种方法既快速，对存储器的使用也很有效率。搜索时间减少了，因为搜索被限制在堆的某个部分，而不是整个堆。存储器利用率得到了改善，因为有一个有趣的事实：对分离空闲链表的简单的首次适配搜索相当于对整个堆的最佳适配搜索。

伙伴系统

伙伴系统（buddy system）是分离匹配的一种特例，其中每个大小类都是 2 的幂。基本的思路是假设一个堆的大小为 2^m 个字，我们为每个块大小 2^k 维护一个分离空闲链表，其中 $0 \le k \le m$。请求块大小向上舍入到最接近的 2 的幂。最开始时，只有一个大小为 2^m 个字的空闲块。

为了分配一个大小为 2^k 的块，我们找到第一个可用的、大小为 2^j 的块，其中 $k \le j \le m$。如果 $j = k$，那么我们就完成了。否则，我们递归地二分这个块，直到 $j = k$。当我们进行这样的分割时，每个剩下的半块（也叫做伙伴），被放置在相应的空闲链表中。要释放一个大小为 2^k 的块，我们继续合并空闲的伙伴。当我们遇到一个已分配的伙伴时，我们就停止合并。

关于伙伴系统的一个关键事实是，给定地址和块的大小，很容易计算出它的伙伴的地址。例如，一个块，大小为 32 字节，地址为：

```
xxx…x00000
```

它的伙伴的地址为

```
xxx…x10000
```

换句话说，一个块的地址和它的伙伴只有一位不相同。

伙伴系统分配器的主要优点是它的快速搜索和快速合并。主要缺点是要求块大小为 2 的幂可能导致显著的内部碎片。因此，伙伴系统分配器不适合通用目的的工作负载。然而，对于某些与应用相关的工作负载，其中块大小预先知道是 2 的幂，伙伴系统分配器就很有吸引力了。

10.10 垃圾收集

在诸如 C malloc 包这样的显式分配器中，应用通过调用 malloc 和 free 来分配和释放堆块。应用要负责释放所有不再需要的已分配块。

未能释放已分配的块是一种常见的编程错误。例如，考虑下面的 C 函数，作为处理的一部分，它分配一块临时存储：

```
1   void garbage()
2   {
3       int *p = (int *)Malloc(15213);
4
5       return; /* array p is garbage at this point */
6   }
```

因为程序不再需要 p，所以在 garbage 返回前应该释放 p。不幸的是，程序员忘了释放这个块。

它在程序的生命周期内都保持为已分配状态，毫无必要地占用着本来可以用来满足后面分配请求的堆空间。

垃圾收集器（garbage collector）是一种动态存储分配器，它自动释放程序不再需要的已分配块。这些块被称为垃圾（garbage），因此术语就称之为垃圾收集器。自动回收堆存储的过程叫做垃圾收集（garbage collection）。在一个支持垃圾收集的系统中，应用显式分配堆块，但是从不显示地释放它们。在 C 程序的上下文中，应用调用 malloc，但是从不调用 free。取而代之的是，垃圾收集器定期识别垃圾块，并相应地调用 free，将这些块放回到空闲链表中。

垃圾收集可以追溯到 John McCarthy 在 20 世纪 60 年代早期在 MIT 开发的 Lisp 系统。它是诸如 Java、ML、Perl 和 Mathematica 等现代语言系统的一个重要部分，而且它仍然是一个重要的研究领域。有关文献描述了大量的垃圾收集方法，其数量令人吃惊。我们的讨论局限于 McCarthy 独创的 Mark&Sweep（标记&清除）算法，这个算法很有趣，因为它可以建立在已存在的 malloc 包的基础之上，为 C 和 C++程序提供垃圾收集。

10.10.1 垃圾收集器的基本要素

垃圾收集器将存储器视为一张有向可达图（reachability graph），其形式如图 10.51 所示。该图的节点被分成一组根节点（root node）和一组堆节点（heap node）。每个堆节点对应于堆中的一个已分配块。有向边 p → q 意味着块 p 中的某个位置指向块 q 中的某个位置。根节点对应于这样一种不在堆中的位置，它们中包含指向堆中的指针。这些位置可以是寄存器，栈里的变量，或者是虚拟存储器中读写数据区域内的全局变量。

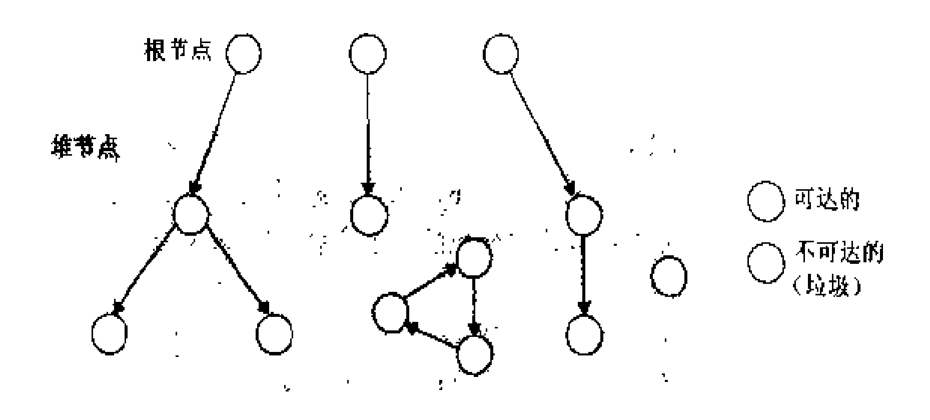


图 10.51 垃圾收集器将存储器视为一张有向图

当存在一条从任意根节点出发并到达 p 的有向路径时，我们说一个节点 p 是可达（reachable）。在任何时刻，和垃圾相对应的不可达节点是不能被应用再次使用的。垃圾收集器的角色是维护可达图的某种表示，并通过释放不可达节点并将它们返回给空闲链表，来定期地回收它们。

像 ML 和 Java 这样的语言的垃圾收集器，对应用如何创建和使用指针有很严格的控制，能够维护可达图的一种精确的表示，因此也就能够回收所有垃圾。然而，诸如 C 和 C++这样的语言的收集器通常不能维持可达图的精确表示。这样的收集器也叫做保守的垃圾收集器（conservative garbage collector）。从某种意义上来说它们是保守的，也就是，每个可达块都被正确地标记为可达了，而一些不可达节点却可能被错误地标记为可达。

收集器可以按需提供它们的服务，或者它们可以作为一个和应用并行的独立线程，不断地更新

可达图和回收垃圾。例如，考虑我们如何为C程序将一个保守的收集器加入到已存在的malloc包中，如图10.52所示。

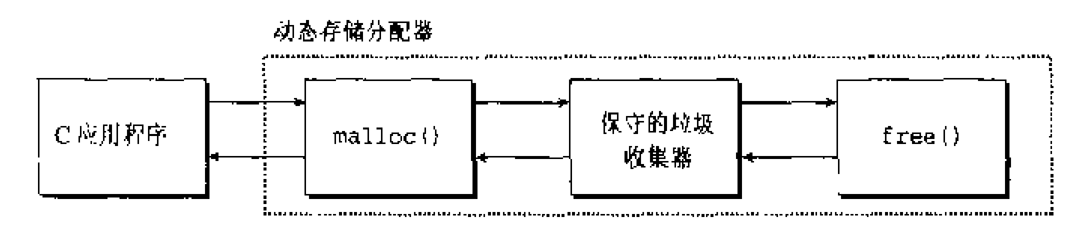


图 10.52 将一个保守的垃圾收集器加入到一个C的malloc包中

无论何时应用需要堆空间时，它都会用通常的方式调用malloc。如果malloc找不到一个合适的空闲块，那么它就调用垃圾收集器，希望能够回收一些垃圾到空闲链表。收集器识别出垃圾块，并通过调用free函数将它们返回给堆。关键的想法是收集器代替应用去调用free。当对收集器的调用返回时，malloc重试，试图发现一个合适的空闲块。如果还是失败了，那么它就会向操作系统要求额外的存储器。最后，malloc返回一个指向请求块的指针（如果成功）或者返回一个空指针（如果不成功）。

10.10.2 Mark&Sweep垃圾收集器

Mark&Sweep垃圾收集器由标记（mark）阶段和清除（sweep）阶段组成。标记阶段标记出根节点的所有可达的和已分配的后继，而后面的清除阶段释放每个未被标记的已分配块。典型地，块头部中空闲的低位中的一位用来表示这个块是否被标记了。

我们对Mark&Sweep的描述将假设使用下列函数，其中ptr定义为typedef char *ptr：

- ptr isPtr (ptr p)：如果p指向一个已分配块中的某个字，那么就返回一个指向这个块的起始位置的指针b。否则返回NULL。
- int blockMarked(ptr b)：如果已经标记了块b，那么就返回true。
- int blockAllocated(ptr b)：如果块b是已分配的，那么就返回true。
- void markBlock(ptr b)：标记块b。
- int length(b)：返回块b的字长（包括头部）。
- void unmarkBlock(ptr b)：将块b的状态由已标记的改为未标记的。
- ptr nextBlock(ptr b)：返回堆中块b的后继。

标记阶段为每个根节点调用一次图10.53（a）所示的mark函数。如果p不指向一个已分配并且未标记的堆块，mark函数就立即返回。否则，它就标记这个块，并对块中的每个字递归地调用它自己。每次对mark函数的调用都标记某个根节点的所有未标记并且可达的后继节点。在标记阶段的末尾，任何未标记的已分配块都被认定为是不可达的，是垃圾，可以在清除阶段回收。

清除阶段是对图10.53（b）所示的sweep函数的一次调用。sweep函数在堆中每个块上反复循环，释放它所遇到的所有未标记的已分配块（也就是垃圾）。

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
```

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
```

```
        return;
    markBlock(b);
    len = length(b);
    for (i=0; i < len; i++)
        mark(b[i]);
    return;
}
```

```
            unmarkBlock(b);
        else if (blockAllocated(b))
            free(b);
        b = nextBlock(b);
    }
    return;
}
```

图 10.53 mark 和 sweep 函数的伪代码

图 10.54 展示了一个小堆的 Mark&Sweep 的图形化解释。块边界用粗线条表示。每个方块对应于存储器中的一个字。每个块有一个字的头部，要么是标记了的，要么是未标记的。

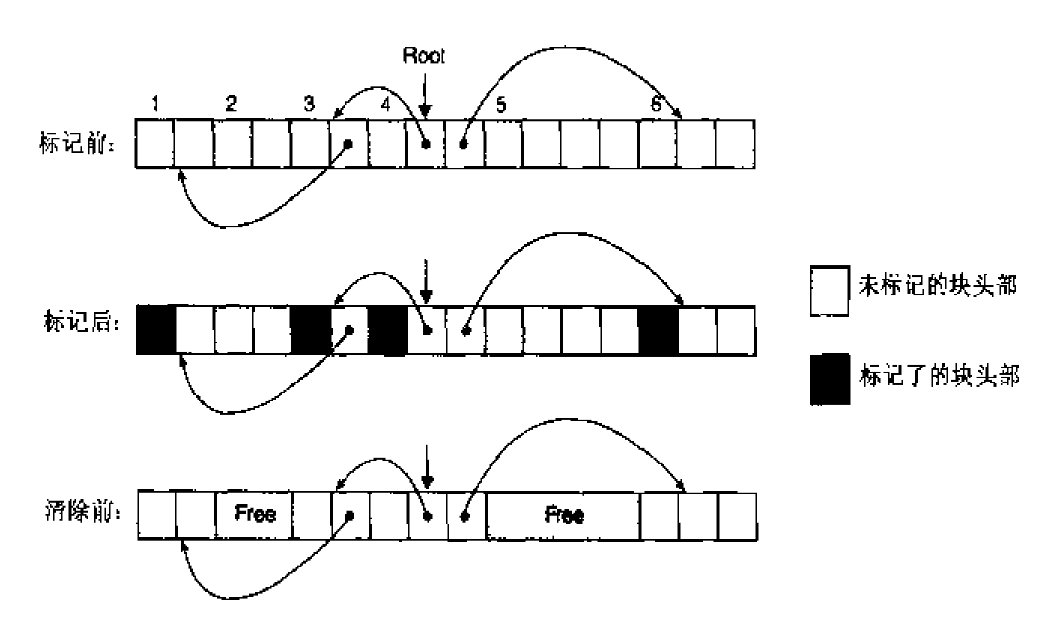


图 10.54 标记和清除示例

注意这个示例中的箭头表示存储器引用，而不是空闲链表指针。

初始情况下，图 10.54 中的堆由六个已分配块组成，其中每个块都是未分配的。第 3 块包含一个指向第 1 块的指针。第 4 块包含指向第 3 块和第 6 块的指针。根指向第 4 块。在标记阶段之后，第 1 块、第 3 块、第 4 块和第 6 块被做了标记，因为它们是从根节点可达的。第 2 块和第 5 块是未标记的，因为它们是不可达的。在清除阶段之后，这两个不可达块被回收到空闲链表。

10.10.3 C 程序的保守 Mark&Sweep

Mark&Sweep 对 C 程序的垃圾收集是一种合适的方法，因为它可以就地工作，而不需要移动任何块。然而，C 语言为 isPtr 函数的实现造成了一些有趣的挑战。

第一，C 不会用任何类型信息来标记存储器位置。因此，对 isPtr 没有一种明显的方式来判断它的输入参数 p 是不是一个指针。第二，即使我们知道 p 是一个指针，对 isPtr 也没有明显的方式来判断 p 是否指向一个已分配块的有效载荷中的某个位置。

对后一问题的解决方法是将已分配块集合维护成一棵平衡二叉树，这棵树保持着这样一个属性：左子树中的所有块都放在较小的地址处，而右子树中的所有块都放在较大的地址处。如图 10.55 所

示，这就要求每个已分配块的头部里有两个附加字段（left 和 right）。每个字段指向某个已分配块的头部。

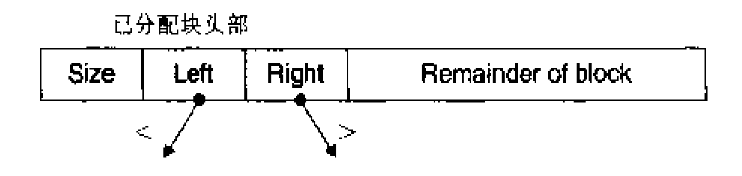


图 10.55 一棵已分配块的平衡树中的左右指针

isPtr(ptr p)函数用树来执行对已分配块的二分查找。在每一步中，它依赖于块头部中的大小字段，来判断 p 是否落在这个块的范围之内。

从某种意义上来说，平衡树方法是正确的，例如它保证会标记所有从根节点可达的节点。这是一个必要的保证，因为应用程序的用户当然不会喜欢把它们的已分配块过早地返回给空闲链表。然而，这种方法从某种意义上而言又是保守的，因为它可能不正确地标记实际上不可达的块，并因此不能释放某些垃圾。虽然这并不影响应用程序的正确性，但是这可能导致不必要的外部碎片。

C 程序的 Mark&Sweep 收集器必须是保守的，其根本原因是 C 语言不会用类型信息来标记存储器位置。因此，像 int 或者 float 这样的标量可以伪装成指针。例如，假设某个可达的已分配块在它的有效载荷中包含一个 int，其值碰巧对应于某个其他已分配块 b 的有效载荷中的一个地址。对收集器而言，是没有办法推断出这个数据实际上是 int 而不是指针。因此，分配器必须保守地将块 b 标记为可达，尽管事实上它可能是不是可达的。

10.11 C 程序中常见的与存储器有关的错误

对 C 程序员来说，管理和使用虚拟存储器可能是个困难的、容易出错的任务。与存储器有关的错误属于那些最令人惊恐的错误，因为它们经常在时间和空间上，都在距错误源一段距离之后，才表现出来。将错误的数据编写到错误的位置，你的程序可能在最终失败之前运行了好几个小时，且使程序中止的位置距离错误的位置已经很远了。我们用一些常见的与存储器有关错误的讨论，来结束我们对虚拟存储器的讨论。

10.11.1 间接引用坏指针

正如我们在 10.7.2 节中学到的，在进程的虚拟地址空间中有较大的洞，没有映射到任何有意义的数据。如果我们试图间接引用一个指向这些洞的指针，那么操作系统就会以段异常终止我们的程序。而且，虚拟存储器的某些区域是只读的。试图写这些区域将造成以保护异常终止这个程序。

间接引用坏指针的一个常见示例是经典的 scanf 错误。假设我们想要使用 scanf 从 stdin 读一个整数到一个变量。做这件事情正确的方法是传递给 scanf 一个格式串和变量的地址：

```
scanf("%d", &val)
```

然而，对于 C 程序员初学者而言（对有经验者也是如此！），很容易传递 val 的内容，而不是它的地址：

```
scanf("%d", val)
```

在这种情况下，scanf 将把 val 的内容解释为一个地址，并试图将一个字写到这个位置。在最好的情况下，程序立即以异常终止。在最糟糕的情况下，val 的内容对应于虚拟存储器的某个合法的读/写区域，于是我们就覆盖了存储器，这通常会在相当以后造成灾难性的、令人困惑的后果。

10.11.2 读未初始化的存储器

虽然.bss 存储器位置（诸如未初始化的全局 C 变量）总是被加载器初始化为零，但是对于堆存储器却并不是这样的。一个常见的错误就是假设堆存储器被初始化为零：

```
1   /* return y = Ax */
2   int *matvec(int **A, int *x, int n)
3   {
4       int i, j;
5
6       int *y = (int *)Malloc(n * sizeof(int));
7
8       for (i = 0; i < n; i++)
9           for (j = 0; j < n; j++)
10              y[i] += A[i][j] * x[j];
11      return y;
12  }
```

在这个示例中，程序员不正确地假设向量 y 被初始化为零。正确的实现方式是在第 8 行和第 9 行之间将 y[i] 设置为零，或者使用 calloc。

10.11.3 允许栈缓冲区溢出

正如我们在 3.13 节中已经看到的，如果一个程序不检查输入串的大小就写入栈中的目标缓冲区，那么这个程序就会有缓冲区溢出错误（buffer overflow bug）。例如，下面的函数就有缓冲区错误，因为 gets 函数拷贝一个任意长度的串到缓冲区。为了纠正这个错误，我们必须使用 fgets 函数，这个函数限制了输入串的大小：

```
1   void bufoverflow()
2   {
3       char buf[64];
4
5       gets(buf); /* here is the stack buffer overflow bug */
6       return;
7   }
```

10.11.4 假设指针和它们指向的对象是相同大小的

一种常见的错误是假设指向对象的指针和它们所指向的对象是相同大小的：

```
1   /* Create an nxm array */
2   int **makeArray1(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int));
6
7       for (i = 0; i < n; i++)
```

```
8            A[i] = (int *)Malloc(m * sizeof(int));
9        return A;
10   }
```

这里的目的是创建一个由 *n* 个指针组成的数组，每个指针都指向一个包含 *m* 个 int 的数组。然而，因为程序员在第 5 行将 sizeof(int *)写成了 sizeof(int)，代码实际创建的是一个 int 的数组。这段代码只有在 int 和指向 int 的指针大小相同的机器上运行良好。

但是，如果我们在像 Alpha 这样的机器上运行这段代码，其中指针大于 int，那么第 7 行和第 8 行的循环将写到超出 A 数组末端的地方。因为这些字中的一个很可能是已分配块的边界标记脚部，所以我们可能不会发现这个错误，直到我们在这个程序的后面很久释放这个块时，此时，分配器中的合并代码会戏剧性地失败，而没有任何明显的原因。这是“在远处起作用（action at distance）”的一个阴险示例，这类“在远处起作用”是与存储器有关的编程错误的典型情况。

10.11.5 造成错位错误

错位（Off-by-one）错误是另一种很常见的覆盖错误发生的原因：

```
1    /* Create an nxm array */
2    int **makeArray2(int n, int m)
3    {
4        int i;
5        int **A = (int **)Malloc(n * sizeof(int));
6
7        for (i = 0; i <= n; i++)
8            A[i] = (int *)Malloc(m * sizeof(int));
9        return A;
10   }
```

这是前面一节中程序的另一个版本。这里我们在第 5 行创建了一个 *n* 个元素的指针数组，但是随后在第 7 行和第 8 行试图初始化这个数组的 *n*+1 个元素，在这个过程中覆盖了 A 数组后面的某个存储器。

10.11.6 引用指针，而不是它所指向的对象

如果我们不太注意 C 操作符的优先级和结合性，我们就会错误地操作指针，而不是期望操作指针所指向的对象。比如，考虑下面的函数，其目的是删除一个有*size 项的二叉堆里的第一项，然后对剩下的*size-1 项重新建堆。

```
1    int *binheapDelete(int **binheap, int *size)
2    {
3        int *packet = binheap[0];
4
5        binheap[0] = binheap[*size - 1];
6        *size--; /* this should be (*size)-- */
7        heapify(binheap, *size, 0);
8        return(packet);
9    }
```

在第 3 行，目的是减少 size 指针指向的整数的值（也就是说是(*size)--）。然而，因为一元--

和* 运算符优先级相同，从右向左结合，所以第6行中的代码实际减少的是指针自己的值，而不是它所指向的整数的值。如果我们幸运地话，程序会立即失败，但是更有可能发生的是，当程序在它执行过程的很后面产生出一个不正确的结果时，我们只能在那里抓破脑袋了。这里的原则是如果你对优先级和结合性有疑问，就使用括号。比如，在第6行，我们可以清晰地表示我们的目的，使用表达式(*size)--。

10.11.7 误解指针运算

另一种常见的错误是忘记了指针的算术操作是以它们指向的对象的大小为单位来进行的，而这种大小单位并不一定是字节。例如，下面函数的目的是扫描一个 int 的数组，并返回一个指针，指向 val 的首次出现：

```
1    int *search(int *p, int val)
2    {
3        while (*p && *p != val)
4            p += sizeof(int); /* should be p++ */
5        return p;
6    }
```

然而，因为每次循环时，第4行都把指针加了4（一个整数的字节数），函数就不正确地扫描数组中每4个整数。

10.11.8 引用不存在的变量

没有太多经验的C程序员不理解栈的规则，有时会引用不再合法的本地变量，如下列所示：

```
1    int *stackref ()
2    {
3        int val;
4
5        return &val;
6    }
```

这个函数返回一个指针（比如说是p），指向栈里的一个局部变量，然后弹出它的栈帧。尽管p仍然指向一个合法的存储器地址，但是它已经不再指向一个合法的变量了。当以后在程序中调用其他函数时，存储器将重用它们的栈帧。后来，如果程序分配某个值给*p，那么它可能实际正在修改另一个函数的栈帧中的一个条目，从而带来潜在地灾难性的、令人困惑的后果。

10.11.9 引用空闲堆块中的数据

一个相似的错误是引用已经被释放了的堆块中的数据。例如，考虑下面的示例，这个示例在第6行分配了一个整数数组x，在第12行先释放了块x，然后在第14行又引用了它。

```
1    int *heapref(int n, int m)
2    {
3        int i;
4        int *x, *y;
5
6        x = (int *)Malloc(n * sizeof(int));
```

```
7
8        /* ... */      /* other calls to malloc and free go here */
9
10       free(x);
11
12       y = (int *)Malloc(m * sizeof(int));
13       for (i = 0; i < m; i++)
14           y[i] = x[i]++;    /* oops! x[i] is a word in a free block */
15
16       return y;
17   }
```

取决于在第6行和第10行发生的malloc和free的调用模式，当程序在第14行引用x[i]时，数组x可能是某个其他已分配堆块的一部分了，因此其内容被重写了。和其他许多与存储器有关的错误一样，这个错误只会在程序执行的后面，当我们注意到y中的值被破坏了时，才会显现出来。

10.11.10 引起存储器泄漏

存储器泄漏是缓慢、隐性的杀手，当程序员不小心忘记释放已分配块，而在堆里创建了垃圾时，会发生这种问题。例如，下面的函数分配了一个堆块x，然后不释放它就返回。

```
1   void leak(int n)
2   {
3       int *x = (int *)Malloc(n * sizeof(int));
4
5       return;    /* x is garbage at this point */
6   }
```

如果leak经常被调用，那么渐渐地，堆里就会充满了垃圾，最糟糕的情况下，会占有整个虚拟地址空间。对于像守护进程和服务器这样的程序来说，存储器泄漏是特别严重的，根据定义这些程序是不会终止的。

10.12 扼要重述一些有关虚拟存储器的关键概念

在这一章里，我们已经看到了虚拟存储器是如何工作的，系统如何用它来实现某些功能，例如加载程序、映射共享库以及为进程提供私有受保护的地址空间。我们还看到了许多应用程序正确或者不正确地使用虚拟存储器的方式。

一个关键的经验教训是，即使虚拟存储器是由系统自动提供的，它也是一种有限的存储器资源，应用程序必须精明地管理它。正如我们从对动态存储分配器的研究中学到的那样，管理虚拟存储器资源可能包括一些微妙的时间和空间的平衡。另一个关键的经验教训是，在C程序中很容易犯与存储器有关的错误。坏的指针值、释放已经空闲了的块、不恰当的强制类型转换和指针运算，以及覆盖堆结构，这些只是可能给我们带来麻烦的许多方式中的一小部分。实际上，与存储器有关的错误很讨厌，这是导致Java产生的一个重要原因，Java取消了取变量地址的能力，完全控制了动态存储分配器，从而严格控制了对虚拟存储器的访问。

10.13 小结

虚拟存储器是对主存的一个抽象。支持虚拟存储器的处理器通过使用一种叫做虚拟寻址的间接形式来引用主存。处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件和软件紧密合作。专门的硬件通过使用页表来翻译虚拟地址，而页表的内容是由操作系统提供的。

虚拟存储器提供三个重要的功能。第一，它在主存中自动缓存最近使用的存放磁盘上的虚拟地址空间的内容。虚拟存储器缓存中的块叫做页。对磁盘上页的引用会触发缺页，缺页将控制转移到操作系统中的一个缺页处理程序。缺页处理程序将页面从磁盘拷贝到主存缓存，如果必要，将写回被驱逐的页。第二，虚拟存储器简化了存储器管理，进而又简化了链接、在进程间共享数据、进程的存储器分配，以及程序加载。最后，虚拟存储器通过在每条页表条目中加入保护位，从而了简化了存储器保护。

地址翻译的过程必须和系统中任意硬件缓存的操作集成在一起。大多数页表条目位于 L1 高速缓存中，但是一个称为 TLB 的页表条目在芯片上的高速缓存，通常会消除访问在 L1 上的页表条目的开销。

现代系统通过将虚拟存储器组块（chunk）和磁盘上的文件组块关联起来，来初始化虚拟存储器组块，这个过程称为存储器映射。存储器映射为共享数据、创建新的进程以及加载程序，提供了一种高效的机制。应用可以使用 mmap 函数来手工地创建和删除虚拟地址空间的区域。然而，大多数程序依赖于动态存储器分配器，例如 malloc，它管理虚拟地址空间区域内一个称为堆的区域。动态存储器分配器是一个有系统级感觉的应用级程序，它直接操作存储器，而无需类型系统的很多帮助。分配器有两种类型：显式分配器要求应用显式地释放它们的存储器块；隐式分配器（垃圾收集器）自动释放任何无用的和不可达的块。

对于 C 程序员来说，管理和使用虚拟存储器是一件困难和容易出错的任务。常见的错误示例包括：间接引用坏指针，读取未初始化的存储器，允许栈缓冲区溢出，假设指针和它们指向的对象大小相同，引用指针而不是它所指向的对象，误解指针运算，引用不存在的变量，以及引起存储器泄漏。

参考文献说明

Kilburn 和他的同事们发表了关于虚拟存储器的第一篇描述[42]。体系结构教科书包括关于硬件在虚拟存储器中的角色的额外细节[33]。操作系统教科书包含关于操作系统角色的额外信息[70，83，75]。

Knuth 在 1968 年编写了有关存储分配的经典之作[43]。从那以后，在这个领域就有了大量的文献。Wilson、Johnstone、Neely 和 Boles 编写了关于显式分配器的完美调查和性能评价的文章[88]。本书中关于各种分配器策略的吞吐率和利用率的一般评价就引自于他们的调查。Jones 和 Lins 提供了关于垃圾收集的全面的调查[37]。Kernighan 和 Ritchie[40]展示了一个简单分配器的完整代码，这个简单的分配器是基于显式空闲链表的，每个空闲块中都有一个块大小和后继指针。这段代码使用联合（union）来消除大量的复杂指针运算，这是很有趣的，但是代价是释放操作是线性时间（而不是常数时间）。

www.cs.colorado.edu/~zorn/DSA.html 上 Zorn 的 Dynamic Storage Allocation Repository（动态存储分配仓库）是一个很方便的资源。它包括检测与存储器相关的错误的调试工具，以及 malloc/free 和垃圾收集器的实现。

家庭作业

10.11 ◆

在下面的一系列问题中，你要展示 10.6.4 节中的示例存储器系统如何将虚拟地址翻译成物理地址，以及如何访问缓存。对于给定的虚拟地址，请指出访问的 TLB 条目、物理地址，以及返回的缓存字节值，请指明是否 TLB 不命中，是否发生了缺页，是否发生了缓存不命中。如果有缓存不命中，对于“返回的缓存字节”用“-”来表示。如果有缺页，对于“PPN”用“-”来表示，并把部分 C 和 D 就空着。

虚拟地址：0x027c

A. 虚拟地址格式

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. 地址翻译

参 数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中？（Y/N）	
缺页？（Y/N）	
PPN	

C. 物理地址格式

11	10	9	8	7	6	5	4	3	2	1	0

D. 物理地址引用

参 数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中？（Y/N）	
返回的缓存字节	

10.12 ◆

对于下面的地址，重复习题 10.11：

虚拟地址：0x03a9

A. 虚拟地址格式

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. 地址翻译

参数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中？（Y/N）	
缺页？（Y/N）	
PPN	

C. 物理地址格式

11	10	9	8	7	6	5	4	3	2	1	0

D. 物理地址引用

参数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中？（Y/N）	
返回的缓存字节	

10.13 ◆

对于下面的地址，重复习题 10.11：

虚拟地址：0x0040

A. 虚拟地址格式

13	12	11	10	9	8	7	6	5	4	3	2	1	0

B. 地址翻译

参 数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中？（Y/N）	
缺页？（Y/N）	
PPN	

C. 物理地址格式

11	10	9	8	7	6	5	4	3	2	1	0

D. 物理地址引用

参 数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中？（Y/N）	
返回的缓存字节	

10.14 ◆◆

假设有一个输入文件 hello.txt，由字符串“Hello, world!\n”组成。编写一个 C 程序，使用 mmap 来改变 hello.txt 的内容为“Jello, world!\n”。

10.15 ◆

确定下面的 malloc 请求序列得到的块大小和头部值。假设：分配器维持双字对齐，使用图 10.37 中块格式的隐式空闲链表；块大小向上舍入为最接近的 8 字节的倍数。

请 求	块大小（十进制字节）	块头部（十六进制）
malloc(3)		
malloc(11)		
malloc(20)		
malloc(21)		

10.16 ◆

确定下面对齐要求和块格式的每个组合的最小块大小。假设：显式空闲链表、每个空闲块中有四字节的 pred 和 succ 指针、不允许有效载荷的大小为零，并且头部和脚部存放在一个四字节的字里。

对齐要求	已分配块	空闲块	最小块大小（字节）
单字	头部和脚部	头部和脚部	
单字	头部，但是没有脚部	头部和脚部	
双字	头部和脚部	头部和脚部	
双字	头部，但是没有脚部	头部和脚部	

10.17 ◆◆◆

开发10.9.12节中的分配器的一个版本，执行下一次适配搜索，而不是首次适配搜索。

10.18 ◆◆◆

10.9.12节中的分配器要求每个块既有头部也有脚部，以实现常数时间的合并。修改分配器，使得空闲块需要头部和脚部，而已分配块只需要头部。

10.19 ◆

下面给出了三组关于存储器管理和垃圾收集的陈述，在每一组中，只有一句陈述是正确的。你的任务就是判断哪一句是正确的。

1. (a) 在一个伙伴系统中，最高可达50%的空间因为内部碎片而被浪费了。
   (b) 首次适配存储器分配算法比最佳适配算法要慢一些（平均而言）。
   (c) 只有当空闲链表按照存储器地址递增排序时，使用边界标记来回收才会快速。
   (d) 伙伴系统只会有内部碎片，而不会有外部碎片。
2. (a) 在按照块大小递减的顺序排序的空闲链表上，使用首次适配算法会导致分配性能很低，但是可以避免外部碎片。
   (b) 对于最佳适配方法，空闲块链表应该按照存储器地址的递增排序。
   (c) 最佳适配方法选择请求段匹配的最大的空闲块。
   (d) 在按照块大小递增的顺序排序的空闲链表上，使用首次适配算法与使用最佳适配算法等价。
3. Mark&Sweep垃圾收集器在下列哪种情况下叫做保守的：
   (a) 它们只有在存储器请求不能被满足时才合并被释放的存储器。
   (b) 它们把一切看起来像指针的东西都当作指针。
   (c) 它们只在用尽存储器时，才执行垃圾收集。
   (d) 它们不释放形成循环链表的存储器块。

10.20 ◆◆◆◆

编写你自己的malloc和free版本，将它的运行时间和空间利用率与标准C库提供的malloc版本进行比较。

练习题答案

练习题10.1答案

这道题让你对不同地址空间的大小有了些了解。曾几何时，一个32位地址空间看上去似乎是不可能的大。但是，现在有些数据库和科学应用需要更大的地址空间，而且你会发现这种趋势会继续。在你的有生之年，你可能会抱怨你的个人电脑上那狭促的64位地址空间！

#地址位（n）	#惟一的地址（N）	最大地址
8	$2^8 = 256$	$2^8-1 = 255$
16	$2^{16} = 64K$	$2^{16}-1 = 64K-1$
32	$2^{32} = 4G$	$2^{32}-1 = 4G-1$
48	$2^{48} = 256T$	$2^{48} = 256T-1$
64	$2^{64} = 16384P$	$2^{64} = 16384P-1$

练习题 10.2 答案

因为每个虚拟页面是 $P = 2^p$ 字节，所以在系统中总共有 $2^n/2^p = 2^{n-p}$ 个可能页面，其中每个都需要一个页表条目（PTE）。

n	$P = 2^p$	#PTE 的
16	4K	16
16	8K	8
32	4K	1M
32	8K	512K

练习题 10.3 答案

为了完全掌握地址翻译，你需要很好地理解这类问题。下面是如何解决第一个子问题：我们有 n=32 个虚拟地址位和 m=24 个物理地址位。页面大小是 P=1KB，这意味着对于 VPO 和 PPO，我们都需要 $\log_2(1K)$=10 位。（回想一下，VPO 和 PPO 是相同的。）剩下的地址位分别是 VPN 和 PPN。

P	#VPN 位	#VPO 位	#PPN 位	#PPO 位
1KB	22	10	14	10
2KB	21	11	13	11
4KB	20	12	12	12
8KB	19	13	11	13

练习题 10.4 答案

做一些这样的手工模拟，能很好地巩固你对地址翻译的理解。你会发现写出地址中的所有的位，然后在不同的位字段上画出方框，例如 VPN、TLBI 等等，这会很有帮助。在这个特殊的练习中，没有任何类型的不命中：TLB 有一份 PTE 的拷贝，而缓存有一份所请求数据字的拷贝。对于命中和不命中的一些不同的组合，请参见习题 10.11、10.12 和 10.13。

```
A. 00 0011 1101 0111
B.      VPN:            0xf
        TLBI:           0x3
        TLBT:           0x3
        TLB 命中?       Y
        缺页?           N
        PPN:            0xd
C.      0011 0101 0111
```

```
D.      CO:              0x3
        CI:              0x5
        CT:              0xd
        高速缓存命中?    Y
        高速缓存字节?    0x1d
```

练习题 10.5 答案

解决这个题目将帮助你很好地理解存储器映射。请自己独立完成这道题。我们没有讨论 open、fstat 或者 write 函数，所以你需要阅读它们的帮助页来看看它们是如何工作的。

code/vm/mmapcopy.c

```
1   #include "csapp.h"
2
3   /*
4    * mmapcopy - uses mmap to copy file fd to stdout
5    */
6   void mmapcopy(int fd, int size)
7   {
8       char *bufp; /* ptr to memory mapped VM area */
9
10      bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
11      Write(1, bufp, size);
12      return;
13  }
14
15  /* mmapcopy driver */
16  int main(int argc, char **argv)
17  {
18      struct stat stat;
19      int fd;
20
21      /* check for required command line argument */
22      if (argc != 2) {
23          printf("usage: %s <filename>\n", argv[0]);
24          exit(0);
25      }
26
27      /* copy the input argument to stdout */
28      fd = Open(argv[1], O_RDONLY, 0);
29      fstat(fd, &stat);
30      mmapcopy(fd, stat.st_size);
31      exit(0);
32  }
```

code/vm/mmapcopy.c

练习题 10.6 答案

这道题触及了一些核心的概念，例如对齐要求、最小块大小以及头部编码。确定块大小的一般方法是，将所请求的有效载荷和头部大小的和舍入到对齐要求（在此例中是 8 字节）最近的整数倍。比如，malloc(1)请求的块大小是 4+1=5，然后舍入到 8。而 malloc(13)请求的块大小是 13+4=17，舍

入到24。

请　求	块大小（十进制字节）	块头部（十六进制）
malloc(1)	8	0x9
malloc(5)	16	0x11
malloc(12)	16	0x11
malloc(13)	24	0x19

练习题 10.7 答案

最小块大小对内部碎片有显著的影响。因此，理解和不同分配器设计和对齐要求相关联的最小块大小是很好的。很有技巧的一部分是，要意识到相同的块可以在不同时刻被分配或者被释放。因此，最小块大小就是最小已分配块大小和最小空闲块大小两者的最大值。例如，在最后一个子问题中，最小的已分配块大小是一个4字节头部和一个1字节有效载荷，舍入到8字节。而最小空闲块的大小是一个4字节的头部和一个4字节的脚部，加起来是8字节，已经是8的倍数，就不需要再舍入了。所以，这个分配器的最小块大小就是8字节。

对齐要求	已分配块	空闲块	最小块大小（字节）
单字	头部和脚部	头部和脚部	12
单字	头部，但是没有脚部	头部和脚部	8
双字	头部和脚部	头部和脚部	16
双字	头部，但是没有脚部	头部和脚部	8

练习题 10.8 答案

这里没有特别的技巧。但是解答此题要求你理解我们简单的隐式链表分配器的剩余部分是如何工作的，是如何操作和遍历块的。

code/vm/malloc.c

```
1   static void *find_fit(size_t asize)
2   {
3       void *bp;
4
5       /* first fit search */
6       for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
7           if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8               return bp;
9           }
10      }
11      return NULL; /* no fit */
12  }
```

code/vm/malloc.c

练习题 10.9 答案

这又是一个帮助你熟悉分配器的热身练习。注意对于这个分配器，最小块大小是16字节。如果分割后剩下的块大于或者等于最小块大小，那么我们就分割这个块（第6～10行）。这里惟一有技巧

的部分是要意识到在移动到下一块之前（第8行），你必须放置新的已分配块（第6行和第7行）。

code/vm/malloc.c

```
1   static void place(void *bp, size_t asize)
2   {
3       size_t csize = GET_SIZE(HDRP(bp));
4
5       if ((csize - asize) >= (DSIZE + OVERHEAD)) {
6           PUT(HDRP(bp), PACK(asize, 1));
7           PUT(FTRP(bp), PACK(asize, 1));
8           bp = NEXT_BLKP(bp);
9           PUT(HDRP(bp), PACK(csize-asize, 0));
10          PUT(FTRP(bp), PACK(csize-asize, 0));
11      }
12      else {
13          PUT(HDRP(bp), PACK(csize, 1));
14          PUT(FTRP(bp), PACK(csize, 1));
15      }
16  }
```

code/vm/malloc.c

练习题 10.10 答案

这里有一个会引起外部碎片的模式：应用对第一个大小类做大量的分配和释放请求，然后对第二个大小类做大量的分配和释放请求，接下来是对第三个大小类做大量的分配和释放请求，以此类推。对于每个大小类，分配器都创建了许多不会被回收的存储器，因为分配器不会合并，也因为应用不会再向这个大小类再次请求块了。

第 3 部分

程序间的交互和通信

我们学习计算机系统到现在，一直假设程序是独立运行的，只包含最小限度的输入和输出。然而，在现实世界里，应用程序利用操作系统提供的服务来与 I/O 设备及其他程序通信。

本书的这一部分将使你了解 Unix 操作系统提供的基本 I/O 服务，以及如何用这些服务来构造应用程序，例如 Web 客户端和服务器，它们是通过 Internet 彼此通信的。你将学习编写诸如 Web 服务器这样的可以同时为多个客户端提供服务的并发程序。

当你学完了这个部分，你将稳健步入权威程序员的行列，能够充分理解计算机系统以及它们对你程序的影响。

CHAPTER 11

系统级 I/O

输入/输出（I/O）是在主存（main memory）和外部设备（例如磁盘驱动器、终端和网络）之间拷贝数据的过程。输入操作是从I/O设备拷贝数据到主存，而输出操作是从主存拷贝数据到I/O设备。

所有语言的运行时系统都提供执行I/O的较高级别的工具。例如，ANSI C提供标准I/O库，包含像printf和scanf这样执行带缓冲的I/O函数。C++语言用它的重载操作符<<（输入）和>>（输出）提供了类似的功能。在Unix系统中，是通过使用由内核提供的系统级Unix I/O函数来实现这些较高级别的I/O函数的。大多数时候，高级别I/O函数工作良好，没有必要直接使用Unix I/O。那么为什么还要麻烦地学习Unix I/O呢？

- 了解Unix I/O将帮助你理解其他的系统概念。I/O是系统操作不可或缺的一部分，因此，我们经常遇到I/O和其他系统概念之间的循环依赖。例如，I/O在进程的创建和执行中扮演着关键的角色。反过来，进程创建又在不同进程间的文件共享中扮演着关键角色。因此，要真正理解I/O，你必须理解进程，反之亦然。在对存储器（memory）结构、结构链接和加载、进程以及虚拟存储器的讨论中，我们已经接触了I/O的某些方面。既然你对这些概念有了比较好的理解，我们就能闭合这个循环，更加深入地研究I/O。
- 有时你除了使用Unix I/O以外别无选择。在某些重要的情况中，使用高级I/O函数不太可能，或者不太合适。例如，标准I/O库没有提供读取文件元数据的方式，这些元数据包括文件大小或文件创建时间。更有甚者，I/O库还存在一些问题，使得用它来进行网络编程非常冒险。

这一章向你介绍Unix I/O和标准I/O的一般概念，并且向你展示在你的C程序中如何可靠地使用它们。除了作为一般性的介绍之外，这一章还为我们随后学习网络编程和并发性奠定坚实的基础。

11.1 Unix I/O

一个Unix文件就是一个m字节的序列

$$B_0, B_1, \cdots, B_k, \cdots, B_{m-1}$$

所有的I/O设备，例如网络、磁盘和终端，都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这种将设备优雅地映射为文件的方式，允许Unix内核引出一个简单、低级的应用接口，称为Unix I/O，这使得所有的输入和输出都能以一种统一且一致的方式来执行：

- 打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个I/O设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录这个打开文件的所有信息，而应用程序只需记住这个描述符。

  Unix shell创建的每个进程开始时都有三个打开的文件：标准输入（描述符为0）、标准输出（描述符为1）和标准错误（描述符为2）。头文件<unistd.h>定义了常量STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO，它们可用来代替显示的描述符值。
- 改变当前的文件位置。内核保持着一个文件位置k，对于每个打开文件，初始为0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行seek操作，显式地设置文件的当前位置为k。
- 读写文件。一个读操作就是从文件拷贝$n>0$个字节到存储器，从当前文件位置k开始，然

后将 k 增加到 $k+n$。给定一个大小为 m 字节的文件，当 $k \geq m$ 时执行读操作会触发一个称为 end-of-file（EOF）的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号”。

类似地，写操作就是从存储器拷贝 $n>0$ 个字节到一个文件，从当前文件位置 k 开始，然后更新 k。

- 关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。内核释放文件打开时创建的数据结构，并恢复描述符到可获得描述符池中，以示响应。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的存储器资源。

11.2 打开和关闭文件

进程是通过调用 open 函数来打开一个已存在的文件或者创建一个新文件的：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(char *filename, int flags, mode_t mode);
```

返回：若成功则为新文件描述符，若出错为-1。

open 函数将 filename 转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。flags 参数指明了进程打算如何访问文件：

- O_RDONLY：只读。
- O_WRONLY：只写。
- O_RDWR：可读可写。

例如，下面的代码说明如何以读的方式打开一个已存在的文件：

```
fd = Open("foo.txt", O_RDONLY, 0);
```

flags 参数也可以是一个或者更多位掩码的或，为写提供给一些额外的指示：

- O_CREAT：如果文件不存在，就创建它的一个截断的（truncated）（空）文件。
- O_TRUNC：如果文件已经存在，就截断它。
- O_APPEND：在每次写操作前，设置文件位置到文件的结尾处。

例如，下面的代码说明的是如何打开一个已存在文件，并在后面添加一些数据：

```
fd = Open("foo.txt", O_WRONLY|O_APPEND, 0);
```

mode 参数指定了新文件的访问权限位。这些位的符号名字如图 11.1 所示。

作为上下文的一部分，每个进程都有一个 umask，它是通过调用 umask 函数来设置的。当进程通过带某个 mode 参数的 open 函数调用来创建一个新文件时，文件的访问权限位被设置为 mode & ~umask。例如，假设我们给定下面的 mode 和 umask 默认值：

```
#define DEF_MODE   S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK  S_IWGRP|S_IWOTH
```

掩 码	描 述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他成员（任何成员）能够读这个文件
S_IWOTH	其他成员（任何成员）能够写这个文件
S_IXOTH	其他成员（任何成员）能够执行这个文件

图 11.1 访问权限位

在 sys/stat.h 中定义。

接下来，下面的代码片段创建一个新文件，文件的拥有者有读写权限，而所有其他的用户都有读权限：

```
umask(DEF_UMASK);
fd = Open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

最后，进程通过调用 close 函数关闭一个打开的文件。

```
#include <unistd.h>

int close(int fd);
```

返回：若成功则为 0，若出错则为 -1。

关闭一个已关闭的描述符会出错。

练习题 11.1

下面程序的输出是什么？

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int fd1, fd2;
6
7       fd1 = Open("foo.txt", O_RDONLY, 0);
8       Close(fd1);
9       fd2 = Open("baz.txt", O_RDONLY, 0);
10      printf("fd2 = %d\n", fd2);
11      exit(0);
12  }
```

11.3 读和写文件

应用程序是通过分别调用 read 和 write 函数来执行输入和输出的。

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
                              返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1。

ssize_t write(int fd, const void *buf, size_t n);
                                        返回：若成功则为写的字节数，若出错则为-1。
```

read 函数从描述符为 fd 的当前文件位置拷贝至多 *n* 个字节到存储器位置 buf。返回值-1 表示一个错误，而返回值 0 表示 EOF。否则，返回值表示的是实际传送的字节数量。

write 函数从存储器位置 buf 拷贝至多 *n* 个字节到描述符 fd 的当前文件位置。图 11.2 展示了一个程序使用 read 和 write 调用一次一个字节地从标准输入拷贝到标准输出。

code/io/cpstdin.c

```
1    #include "csapp.h"
2
3    int main(void)
4    {
5        char c;
6
7        while(Read(STDIN_FILENO, &c, 1) != 0)
8            Write(STDOUT_FILENO, &c, 1);
9        exit(0);
10   }
```

code/io/cpstdin.c

图 11.2 一次一个字节地从标准输入拷贝到标准输出

通过调用 lseek 函数，应用程序能够显示地修改当前文件的位置，这部分内容不在我们的讲述范围之内。

旁注：ssize_t 和 size_t 有些什么区别？

你可能已经注意到了，read 函数有一个 size_t 的输入参数和一个 ssize_t 的返回值。那么这两种类型之间有什么区别呢？size_t 被定义为 unsigned int，而 ssize_t（有符号的大小）被定义为 int。read 函数返回一个有符号的大小，而不是一个无符号大小，这是因为出错时它必须返回-1。有趣的是，返回一个-1 的可能性使得 read 的最大值减小一半，从 4GB 减小到了 2GB。

在某些情况下，read 和 write 传送的字节比应用程序要求的要少。这些**不足值**（short count）不一定是错误。因为一些原因，会出现这样的情况：

- 读时遇到 EOF。假设我们准备读一个文件，该文件从当前文件位置开始只含有 20 多个字节，而我们以 50 个字节的组块（chunk）进行读取。这样一来，下一个 read 返回的不足值为 20，

此后的 read 将通过返回 0 发出 EOF 信号。

- 从终端读文本行。如果打开文件是与终端相关联的（例如，键盘和显示器），那么每个 read 函数将一次传送一个文本行，返回的不足值等于文本行的大小。
- 读和写网络套接字（socket）。如果打开的文件对应于网络套接字（12.3.3 节），那么内部缓冲约束和较长的网络延迟会引起 read 和 write 返回不足值。对 Unix 管道（pipe）调用 read 和 write，也有可能出现不足值，这种进程间通信机制不在我们讨论的范围之内。

实际上，除了 EOF，当你在读磁盘文件时，将不会遇到不足值，而且在写磁盘文件时，也不会遇到不足值。然而，如果你想创建健壮的（可靠的）诸如 Web 服务器这样的网络应用，就必须处理由于反复调用 read 和 write 引起的不足值，直到所有需要的字节都传送完毕。

11.4 用 Rio 包进行健壮地读和写

在这一小节里，我们会讲述一个 I/O 包，称为 Rio（Robust I/O，健壮的 I/O）包，它会自动为你处理上文中所述的不足值。在像网络程序这样容易出现不足值的应用中，Rio 包提供了方便、健壮和高效的 I/O。Rio 提供了两类不同的函数：

- 无缓冲的输入输出函数。这些函数直接在存储器和文件之间传送数据，没有应用级缓冲。它们对将二进制数据读写到网络和从网络读写二进制数据尤其有用。
- 带缓冲的输入函数。这些函数允许你高效地从文件中读取文本行和二进制数据，这些文件的内容缓存在应用级缓冲区内，相似于为 printf 这样的标准 I/O 函数提供的缓冲区。与[81]中讲述的带缓冲的 I/O 例程不同，带缓冲的 Rio 输入函数是线程安全的（13.7.1 节），它在同一个描述符上可以被交错地调用。例如，你可以从一个描述符中读一些文本行，然后读取一些二进制数据，接着再多读取一些文本行。

我们提出 Rio 例程为了两个原因：第一，在接下来的两章中，我们开发的网络应用中使用了它们；第二，通过学习这些例程的代码，你将对 Unix I/O 有更深入的了解。

11.4.1 Rio 的无缓冲的输入输出函数

通过调用 rio_readn 和 rio_writen 函数，应用程序可以在存储器和文件之间直接传送数据。

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

返回：若成功则为传送的字节数，若 EOF 则为 0（只对 rio_readn 而言），若出错则为-1。

rio_readn 函数从描述符 fd 的当前文件位置最多传送 n 个字节到存储器位置 usrbuf。类似地，rio_writen 函数从位置 usrbuf 传送 n 个字节到描述符 fd。rio_read 函数在遇到 EOF 时只能返回一个不足值。rio_writen 函数决不会返回不足值。对同一个描述符，可以任意交错地调用 rio_readn 和 rio_writen。

图 11.3 显示了 rio_readn 和 rio_writen 的代码。注意，如果 read 和 write 函数被一个从应用程序信号处理程序的返回中断，那么每个函数都会手动地重启 read 或 write。为了尽可能有较好的可移

植性，我们允许被中断的系统调用，且在必要时重启它们。（参见8.5.4节中关于被中断的系统调用的讨论。）

code/src/csapp.c

```
1   ssize_t rio_readn(int fd, void *usrbuf, size_t n)
2   {
3       size_t nleft = n;
4       ssize_t nread;
5       char *bufp = usrbuf;
6
7       while (nleft > 0) {
8           if ((nread = read(fd, bufp, nleft)) < 0) {
9               if (errno == EINTR)  /* interrupted by sig handler return */
10                  nread = 0;       /* and call read() again */
11              else
12                  return -1;       /* errno set by read() */
13          }
14          else if (nread == 0)
15              break;               /* EOF */
16          nleft -= nread;
17          bufp += nread;
18      }
19      return (n - nleft);          /* return >= 0 */
20  }
```

code/src/csapp.c

code/src/csapp.c

```
1   ssize_t rio_writen(int fd, void *usrbuf, size_t n)
2   {
3       size_t nleft = n;
4       ssize_t nwritten;
5       char *bufp = usrbuf;
6
7       while (nleft > 0) {
8           if ((nwritten = write(fd, bufp, nleft)) <= 0) {
9               if (errno == EINTR)  /* interrupted by sig handler return */
10                  nwritten = 0;    /* and call write() again */
11              else
12                  return -1;       /* errorno set by write() */
13          }
14          nleft -= nwritten;
15          bufp += nwritten;
16      }
17      return n;
18  }
```

code/src/csapp.c

图 11.3 rio_readn 和 rio_writen 函数

11.4.2 Rio 的带缓冲的输入函数

一个文本行就是一个由换行符结尾的 ASCII 码字符序列。在 Unix 系统中，换行符（"\n"）与 ASCII 码换行符（LF）相同，数字值为 0x0a。假设我们要编写一个程序来计算文本文件中文本行的数量。我们该如何来实现呢？一种方法就是用 read 函数来一次一个字节地从文件传送到用户存储器，检查每个字节来查找换行符。这个方法的缺点是它的效率不是很高，每读取文件中的一个字节都要求陷入内核。

一种更好的方法是调用包装（wrapper）函数（rio_readlineb），它从一个内部读缓冲区拷贝一个文本行，当缓冲区变空时，会自动地调用 read 重新填满缓冲区。对于既包含文本行也包含二进制数据的文件（例如 12.5.3 节中描述的 HTTP 响应），我们也提供了一个 rio_readn 带缓冲区的版本，叫做 rio_readnb，它从和 rio_readlineb 一样的缓冲区中传送原始字节。

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
```

返回：无。

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

返回：若成功则为读的字节数，若 EOF 则为 0，若出错则为-1。

每打开一个描述符，都会调用 rio_readinitb 函数。它将描述符 fd 和地址 rp 处的一个类型为 rio_t 的读缓冲区联系起来。

rio_readinitb 函数从文件 rp 读出一个文本行（包括结尾的换行符），将它拷贝到存储器位置 usrbuf，并且用 null（空）字符来结束这个文本行。rio_readinitb 函数最多读 maxlen-1 个字节，余下的一个字符留给结尾的空字符。超过 maxlen-1 字节的文本行被截断，并用一个空字符结束。

rio_readnb 函数从文件 rp 最多读 n 个字节到存储器位置 usrbuf，对同一描述符，对 rio_readlineb 和 rio_readn 的调用可以任意交叉进行。然而，对这些带缓冲的函数的调用却不应和不带缓冲的 rio_readn 函数交叉使用。

你将在本书剩下的部分中遇到大量的 Rio 函数的示例。图 11.4 展示了如何使用 Rio 函数来一次一行地从标准输入拷贝一个文本文件到标准输出。

code/io/cpfile.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       int n;
6       rio_t rio;
7       char buf[MAXLINE];
8
9       Rio_readinitb(&rio, STDIN_FILENO);
10      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
```

```
11          Rio_writen(STDOUT_FILENO, buf, n);
12  }
```

code/io/cpfile.c

图 11.4 从标准输入拷贝一个文本文件到标准输出

图 11.5 展示了一个读缓冲区的格式，以及初始化它的 rio_readinitb 函数的代码。rio_readinitb 函数创建了一个空的读缓冲区，并且将一个打开的文件描述符和这个缓冲区联系起来。

Rio读程序的核心是图11.6所示为rio_read函数。rio_read函数是Unix read函数的带缓冲的版本。

code/include/csapp.h

```
1   #define RIO_BUFSIZE 8192
2   typedef struct {
3       int rio_fd;                  /* descriptor for this internal buf */
4       int rio_cnt;                 /* unread bytes in internal buf */
5       char *rio_bufptr;            /* next unread byte in internal buf */
6       char rio_buf[RIO_BUFSIZE];   /* internal buffer */
7   } rio_t;
```

code/ include /csapp.h

code/src/csapp.c

```
1   void rio_readinitb(rio_t *rp, int fd)
2   {
3      rp->rio_fd = fd;
4      rp->rio_cnt = 0;
5      rp->rio_bufptr = rp->rio_buf;
6   }
```

code/src/csapp.c

图 11.5 一个类型为 rio_t 的读缓冲区和初始化它的 rio_readinitb 函数

code/src/csapp.c

```
1   static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
2   {
3       int cnt;
4
5       while (rp->rio_cnt <= 0) { /* refill if buf is empty */
6           rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
7                              sizeof(rp->rio_buf));
8           if (rp->rio_cnt < 0) {
9               if (errno != EINTR) /* interrupted by sig handler return */
10                  return -1;
11          }
12          else if (rp->rio_cnt == 0) /* EOF */
13              return 0;
14          else
```

```
15                rp->rio_bufptr = rp->rio_buf; /* reset buffer ptr */
16        }
17
18        /* Copy min(n, rp->rio_cnt) bytes from internal buf to user buf */
19        cnt = n;
20        if (rp->rio_cnt < n)
21            cnt = rp->rio_cnt;
22        memcpy(usrbuf, rp->rio_bufptr, cnt);
23        rp->rio_bufptr += cnt;
24        rp->rio_cnt -= cnt;
25        return cnt;
26    }
```

code/src/csapp.c

图 11.6 内部的 rio_read 函数

当调用 rio_read 要求读 n 个字节时，读缓冲区内有 rp->rio_cnt 个未读字节。如果缓冲区为空，那么会通过调用 read 再次填充它。这个 read 调用收到一个不足值并不是错误，只不过读缓冲区是部分填充的。一旦缓冲区非空，rio_read 就从读缓冲区拷贝 n 和 rp->rio_cnt 中最小值的字节到用户缓冲区，并返回拷贝的字节数。

对于一个应用程序，rio_read 函数和 Unix read 函数有同样的语义。在出错时，它返回值-1，并且适当地设置 errno。在 EOF 时，它返回值 0。如果要求的字节数超过了读缓冲区内未读的字节的数量，它会返回一个不足值。两个函数的相似性使得很容易通过用 rio_read 代替 read 来创建不同类型的带缓冲的读函数。例如，用 rio_read 代替 read，图 11.7 中的 rio_readnb 函数和 rio_readn 有相同的结构。相似地，图 11.7 中的 rio_readlineb 程序最多调用 rio_read maxlen-1 次。每次调用都从读缓冲区返回一个字节，然后会检查这个字节是否是结尾的换行符。

code/src/csapp.c

```
1   ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
2   {
3       int n, rc;
4       char c, *bufp = usrbuf;
5
6       for (n = 1; n < maxlen; n++) {
7           if ((rc = rio_read(rp, &c, 1)) == 1) {
8               *bufp++ = c;
9               if (c == '\n')
10                  break;
11          } else if (rc == 0) {
12              if (n == 1)
13                  return 0;    /* EOF, no data read */
14              else
15                  break;       /* EOF, some data was read */
16          } else
```

```
17              return -1;        /* error */
18          }
19          *bufp = 0;
20          return n;
21      }
```

code/src/csapp.c

code/src/csapp.c

```
1   ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2   {
3       size_t nleft = n;
4       ssize_t nread;
5       char *bufp = usrbuf;
6
7       while (nleft > 0) {
8           if ((nread = rio_read(rp, bufp, nleft)) < 0) {
9               if (errno == EINTR) /* interrupted by sig handler return */
10                  nread = 0;      /* call read() again */
11              else
12                  return -1;      /* errno set by read() */
13              }
14              else if (nread == 0)
15                  break;          /* EOF */
16              nleft -= nread;
17              bufp += nread;
18      }
19      return (n - nleft);         /* return >= 0 */
20  }
```

code/src/csapp.c

图 11.7 rio_readlineb 和 rio_readnb 函数

旁注：Rio 包的起源

Rio 函数的灵感来自于 W. Richard Stevens 在他的经典网络编程作品[81]中描述的 readline、readn 和 writen 函数。rio_readn 和 rio_writen 函数与 Stevens 的 readn 和 writen 函数是一样的。然而，Stevens 的 readline 函数有一些限制在 Rio 中得到了纠正。第一，因为 readline 是带缓冲的，而 readn 不带，所以这两个函数不能在同一描述符上一起使用。第二，因为它使用一个静态的缓冲区，Stevens 的 readline 函数不是线程安全的，这就要求 Stevens 引入一个不同的线程安全版本，称为 readline_r。我们已经在 rio_readlineb 和 rio_readn 函数中修改了这两个缺陷，使得这两个函数是相互兼容和线程级安全的。

11.5 读取文件元数据

应用程序能够通过调用 stat 和 fstat 函数，检索到关于文件的信息（有时也称为文件的元数据，metadata）。

```
#include <unistd.h>
#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

返回：若成功则为0，若出错则为-1。

stat函数以一个文件名作为输入，并填写如图11.8所示的一个stat数据结构中的各个成员。fstat函数是相似的，只不过是以文件描述符而不是文件名作为输入。当我们在12.5节中讨论Web服务器时，会需要stat数据结构中的st_mode和st_size成员，其他成员则不在我们的讨论之列。

statbuf.h(includeed by sys/stat.h)

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

statbuf.h(includeed by sys/stat.h)

图11.8 stat数据结构

st_size成员包含了文件的字节数大小。st_mode成员则编码了文件访问许可位（图11.1）和文件类型。Unix识别大量不同的文件类型。普通文件包括某种类型的二进制或文本数据。对于内核而言，文本文件和二进制文件毫无区别。目录文件包含关于其他文件的信息。套接字是一种用来通过网络与其他进程通信的文件。

Unix提供的宏指令根据st_mode成员来确定文件的类型。图11.9列出了这些宏的一个子集。

宏指令	描述
S_ISREG()	这是一个普通文件吗？
S_ISDIR()	这是一个目录文件吗？
S_ISSOCK()	这是一个网络套接字吗？

图11.9 根据st_mode位确定文件类型的宏指令

在sys/stat.h中定义。

图 11.10 展示了我们会如何使用这些宏和 stat 函数，来读取和解释一个文件的 st_mode 位。

code/io/statcheck.c

```
1   #include "csapp.h"
2
3   int main (int argc, char **argv)
4   {
5       struct stat stat;
6       char *type, *readok;
7
8       Stat(argv[1], &stat);
9       if (S_ISREG(stat.st_mode))        /* Determine file type */
10          type = "regular";
11      else if (S_ISDIR(stat.st_mode))
12          type = "directory";
13      else
14          type = "other";
15      if ((stat.st_mode & S_IRUSR))   /* Check read access */
16          readok = "yes";
17      else
18          readok = "no";
19
20      printf("type: %s, read: %s\n", type, readok);
21      exit(0);
22  }
```

code/io/statcheck.c

图 11.10 查询和处理一个文件的 st_mode 位

11.6 共享文件

可以用许多不同的方式来共享 Unix 文件。除非你很清楚内核是如何表示打开的文件，否则文件共享的概念相当难懂。内核用三种相关的数据结构来表示打开的文件：

- 描述符表（descriptor table）。每个进程都有它独立的描述符表，它的表项是由进程打开的文件描述符来索引的。每个打开的描述符表项指向文件表中的一个表项。
- 文件表（file table）。打开文件的集合是由一张文件表来表示的，所有的进程共享这张表。每个文件表的表项组成（针对我们的目的）包括有当前的文件位置、引用计数（reference count）即当前指向该表项的描述符表项数，以及一个指向 v-node 表中对应表项的指针。关闭一个描述符会减少相应的文件表表项中的引用计数。内核不会删除这个文件表表项，直到它的引用计数为零。
- v-node 表（v-node table）。同文件表一样，所有的进程共享这张 v-node 表。每个表项包含

stat 结构中的大多数信息，包括 st_mode 和 st_size 成员。

图 11.11 展示了一个示例，其中描述符 1 和 4 通过不同的打开文件表表项来引用两个不同的文件。这是一种典型的情况，没有共享文件，并且每个描述符对应一个不同的文件。

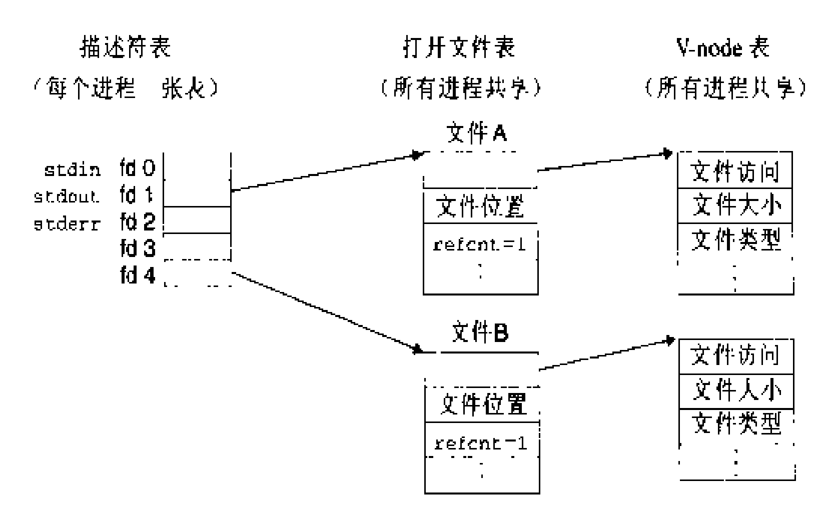


图 11.11 典型的打开文件的内核数据结构

在这个示例中，两个描述符引用不同的文件。没有共享。

如图 11.12 所示，多个描述符也可以通过不同的文件表表项来引用同一个文件。例如，如果以同一个文件名调用 open 函数两次，就会发生这种情况。关键思想是每个描述符都有它自己的文件位置，所以对不同描述符的读操作可以从文件的不同位置获取数据。

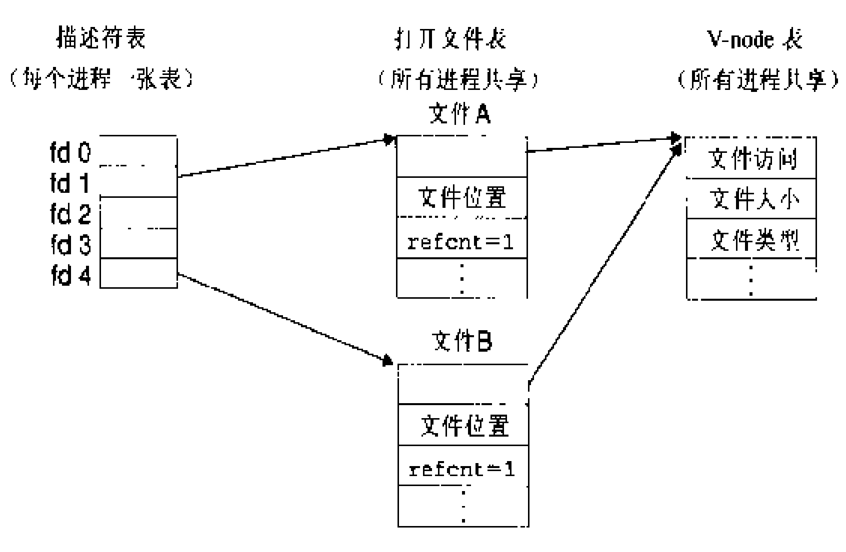


图 11.12 文件共享

这个例子展示了两个描述符通过两个打开文件表表项共享同一个磁盘文件。

我们也能理解父子进程是如何共享文件的。假设在调用 fork 之前，父进程有如图 11.11 所示的打开文件。这时，图 11.13 展示了调用 fork 后的情况。

子进程有一个父进程描述符表的副本。父子进程共享相同的打开文件表集合，因此共享相同的文件位置。一个很重要的结果就是，在内核删除相应文件表表项之前，父子进程必须都关闭了它们的描述符。

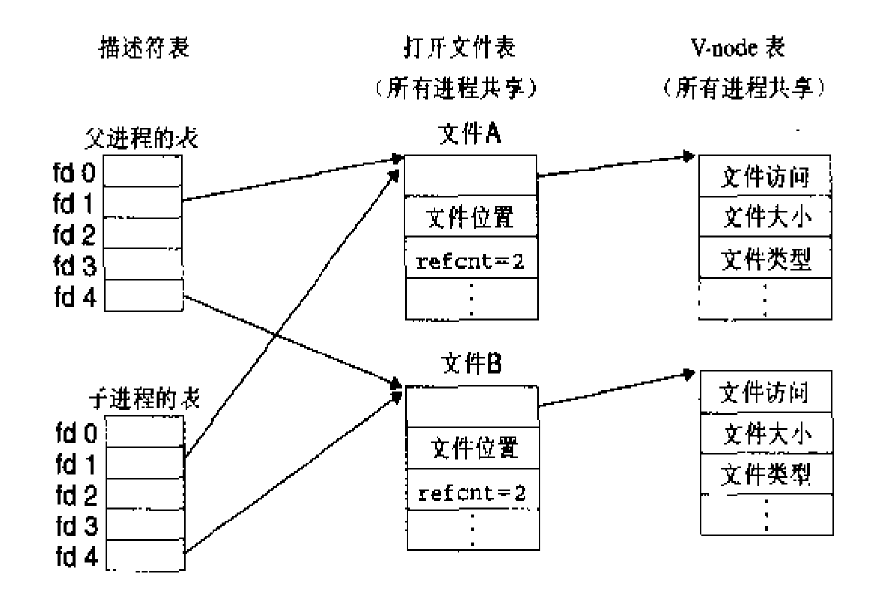


图 11.13　子进程如何继承父进程的打开文件

初始状态如图 11.11 所示。

练习题 11.2

假设磁盘文件 foobar.txt 由 6 个 ASCII 码字符 "foobar" 组成。那么，下列程序的输出是什么？

```
1    #include "csapp.h"
2
3    int main()
4    {
5        int fd1, fd2;
6        char c;
7
8        fd1 = Open("foobar.txt", O_RDONLY, 0);
9        fd2 = Open("foobar.txt", O_RDONLY, 0);
10       Read(fd1, &c, 1);
11       Read(fd2, &c, 1);
12       printf("c = %c\n", c);
13       exit(0);
14   }
```

练习题 11.3

就像前面那样，假设磁盘文件 foobar.txt 由 6 个 ASCII 码字符 "foobar" 组成。那么下列程序的输出是什么？

```
1    #include "csapp.h"
2
3    int main()
4    {
```

```
5       int fd;
6       char c;
7
8       fd = Open("foobar.txt", O_RDONLY, 0);
9       if (Fork() == 0) {
10          Read(fd, &c, 1);
11          exit(0);
12      }
13      Wait(NULL);
14      Read(fd, &c, 1);
15      printf("c = %c\n", c);
16      exit(0);
17  }
```

11.7 I/O重定向

Unix shell提供了I/O重定向操作符，允许用户将磁盘文件和标准输入输出联系起来。例如，键入

```
Unix> ls > foo.txt
```

使得shell加载和执行ls程序，将标准输出重定向到磁盘文件foo.txt。就如我们将在12.5节中看到的那样，当一个Web服务器代表客户端运行CGI程序时，它就执行一种相似类型的重定向。那么I/O重定向是如何工作的呢？一种方式是使用dup2函数。

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

返回：若成功则为非负的描述符，若出错则为-1。

dup2函数拷贝描述符表表项oldfd到描述符表表项newfd，覆盖描述符表表项newfd以前的内容。如果newfd已经打开了，dup2会在拷贝oldfd之前关闭newfd。

假设在调用dup2(4,1)之前，我们的状态如图11.11所示，其中描述符1（标准输出）对应于文件A（比如说一个终端），描述符4对应于文件B（比如说一个磁盘文件）。A和B的引用计数都等于1。图11.14显示了调用dup2(4,1)之后的情况。两个描述符现在都指向B；文件A已经被关闭了，并且它的文件表和v-node表表项也已经删除了；文件B的引用计数已经增加了。从此以后，任何写到标准输出的数据都被重定向到文件B。

旁注：左边和右边的hoinkies。

为了避免和其他括号类型操作符比如“]”和“[”相混淆，我们总是将外壳的“>”操作符称为“右hoinky”，而将“<”操作符称为“左hoinky”。

练习题 11.4

如何用dup2将标准输入重定向到描述符5？

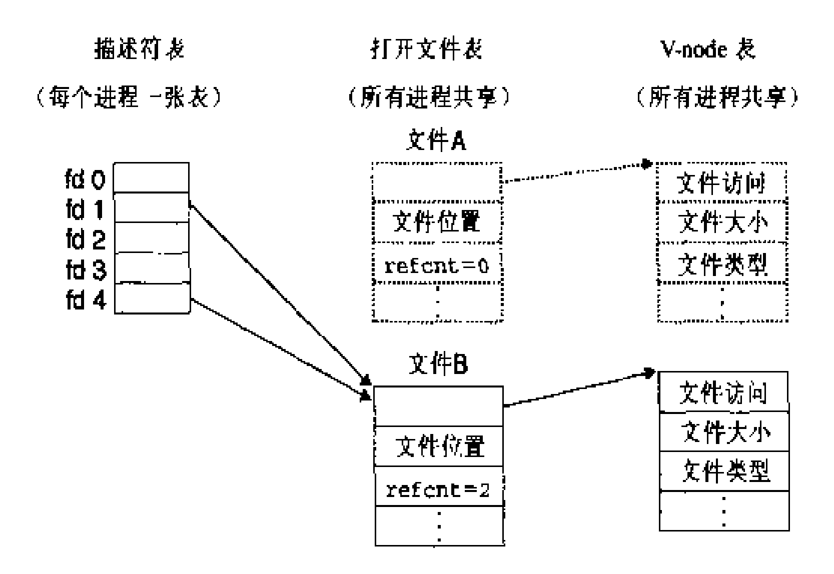


图 11.14 通过调用 dup2(4,1)重定向标准输出之后的内核数据结构

初始状态如图 11.11 所示。

练习题 11.5

假设磁盘文件 foobar.txt 由 6 个 ASCII 码字符"foobar"组成，那么下列程序的输出是什么？

```
1    #include "csapp.h"
2
3    int main()
4    {
5        int fd1, fd2;
6        char c;
7
8        fd1 = Open("foobar.txt", O_RDONLY, 0);
9        fd2 = Open("foobar.txt", O_RDONLY, 0);
10       Read(fd2, &c, 1);
11       Dup2(fd2, fd1);
12       Read(fd1, &c, 1);
13       printf("c = %c\n", c);
14       exit(0);
15   }
```

11.8 标准 I/O

ANSI C 定义了一组高级输入输出函数，称为标准 I/O 库，为程序员提供了 Unix I/O 的较高级别的接口。这个库（libc）提供了打开和关闭文件的函数（fopen 和 fclose）、读和写字节的函数（fread 和 fwrite）、读和写字符串的函数（fgets 和 fputs），以及复杂的格式化 I/O 函数（scanf 和 printf）。

标准 I/O 库将一个打开的文件模型化为一个流。对于程序员而言，一个流就是一个指向类型为 FILE 结构的指针。每个 ANSI C 程序开始时都有三个打开的流 stdin、stdout 和 stderr，分别对应于标准输入、标准输出和标准错误：

```
#include <stdio.h>
extern FILE *stdin;     /* standard input (descriptor 0) */
extern FILE *stdout;    /* standard output (descriptor 1) */
extern FILE *stderr;    /* standard error (descriptor 2) */
```

类型为 FILE 的流是对文件描述符和流缓冲区的抽象。流缓冲区的目的和 Rio 读缓冲区的一样：就是使开销较高的 Unix I/O 系统调用的数量尽可能得小。例如，假设我们有一个程序，它反复调用标准 I/O 的 getc 函数，每次调用返回文件的下一个字符。当第一次调用 getc 时，库通过调用一次 read 函数来填充流缓冲区，然后将缓冲区中的第一个字节返回给应用程序。只要缓冲区中还有未读的字节，接下来对 getc 的调用就能直接从流缓冲区得到服务。

11.9 综合：我该使用哪些 I/O 函数？

图 11.15 总结了我们在这一章里讨论过的各种 I/O 包。

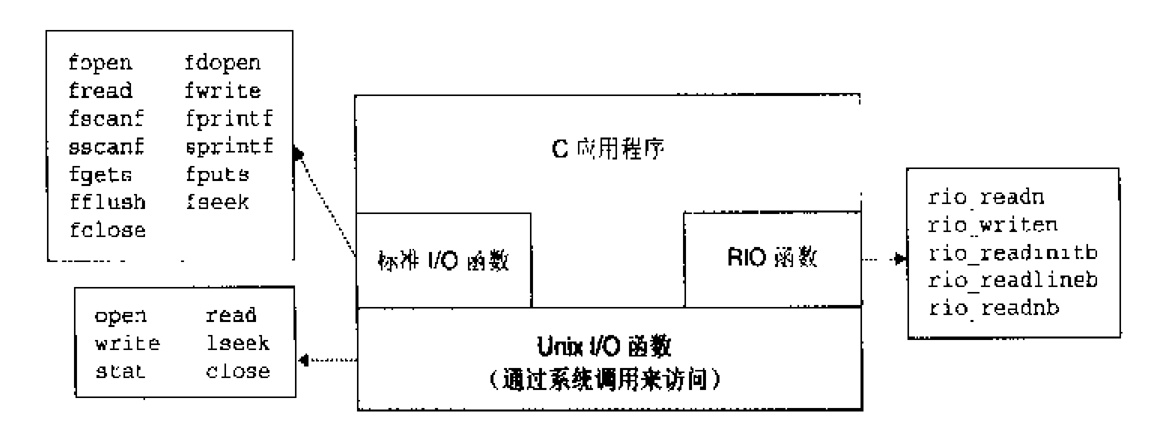


图 11.15 Unix I/O、标准 I/O 和 Rio 之间的关系

Unix I/O 是在操作系统内核中实现的。应用程序可以通过 open、close、lseek、read 和 write 这样的函数来访问 Unix I/O。较高级别的 Rio 和标准 I/O 函数都是基于（使用）Unix I/O 函数来实现的。Rio 函数是专为本书开发的 read 和 write 的健壮包装（wrapper）函数。它们自动处理不足值（short counts），并且为读文本行提供一种高效的带缓冲的方法。标准 I/O 函数提供了 Unix I/O 函数的一个更加完整的带缓冲的替代品，包括格式化 I/O 例程。

那么，在你的程序中该使用这些函数中的哪一个呢？标准 I/O 函数是磁盘和终端设备 I/O 之选。大多数 C 程序员在他们的职业生涯中只使用标准 I/O，而从不涉及低级 Unix I/O 函数。只要可能，我们推荐你也这样做。

不幸的是，当我们试图对网络输入输出使用标准 I/O 时，它却带来了一些令人讨厌的问题。就像我们将在 12.4 节中看到的那样，Unix 对网络的抽象是一种称为套接字的文件类型。和任何 Unix 文件一样，套接字也是用文件描述符来引用的，在这种情况中被称为套接字描述符。应用进程通过读写套接字描述符来与运行在其他计算机上的进程通信。

标准 I/O 流，从某种意义上而言是全双工的，因为程序能够在同一个流上执行输入和输出。然而，很少有文字记载和套接字限定相关的流限定：

- 限定一：输入函数跟在输出函数之后。如果中间没有插入对 fflush、fseek、fsetpos 或者 rewind

的调用，一个输入函数不能跟随在一个输出函数之后。fflush 函数清空与流相关的缓冲区。后三个函数使用 Unix I/O lseek 函数来重置当前的文件位置。

- 限定二：输出函数跟在输入函数之后。如果中间没有插入对 fflush、fseek、fsetpos 或者 rewind 的调用，一个输出函数不能跟随在一个输入函数之后，除非该输入函数遇到了一个文件结束。

这些限制给网络应用带来了一个问题，因为对套接字使用 lseek 函数是非法的。对流 I/O 的第一个限定能够通过采用在每个输入操作前刷新缓冲区这样的规则来保证实现。然而，保证实现第二个限定的惟一办法是，对同一个打开的套接字描述符打开两个流，一个用来读，一个用来写：

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

但是这种方法也有问题，因为它要求应用程序在两个流上都要调用 fclose，这样才能释放与每个流相关联的存储器资源，避免存储器泄漏：

```
fclose(fpin);
fclose(fpout);
```

这些操作中的每一个都试图关闭同一个底层的套接字描述符，所以第二个 close 操作就会失败。对顺序的程序来说，这并不是问题，但是在一个线程化的（threaded）程序中关闭一个已经关闭了的描述符是会导致灾难的（参见 13.7.4 节）。

因此，我们推荐你在网络套接字上不要使用标准 I/O 函数来进行输入和输出，而要使用 Rio 函数。如果你需要格式化输出，使用 sprintf 函数在存储器中格式化一个字符串，然后用 rio_writen 把它发送到套接口。如果你需要格式化输入，使用 rio_readlineb 来读一个完整的文本行，然后用 scanf 从文本行提取不同的字段。

11.10 小结

Unix 提供了少量的系统级函数，它们允许应用程序打开、关闭、读和写文件，提取文件的元数据，以及执行 I/O 重定向。Unix 的读和写操作会出现不足值（short counts），应用程序必须能正确地预计和处理这种情况。应用程序不直接调用 Unix I/O 函数，而应该使用 Rio 包，Rio 包通过反复执行读写操作，直到传送完所有的请求数据，自动处理不足值。

Unix 内核使用三种相关的数据结构来表示打开的文件。描述符表中的表项指向打开文件表中的表项，而打开文件表中的表项又指向 v-node 表中的表项。每个进程都有它自己单独的描述符表，而所有的进程共享同一打开文件表和 v-node 表。理解这些结构的一般构成就能使我们清楚地理解文件共享和 I/O 重定向。

标准 I/O 库是基于 Unix I/O 实现的，并提供了一组强大的高级 I/O 例程。对于大多数应用程序而言，标准 I/O 更简单，是优于 Unix I/O 的选择。然而，因为对标准 I/O 和网络文件的一些相互不兼容的限制，Unix I/O 比之标准 I/O 更该适用于网络应用程序。

参考文献说明

Stevens 编写了 Unix I/O 的标准参考文献[76]。Kernighan 和 Ritchie 对于标准 I/O 函数给出了清晰而完整的讨论[40]。

家庭作业

11.6 ◆

下面题目的输出是什么？

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int fd1, fd2;
6
7       fd1 = Open("foo.txt", O_RDONLY, 0);
8       fd2 = Open("bar.txt", O_RDONLY, 0);
9       Close(fd2);
10      fd2 = Open("baz.txt", O_RDONLY, 0);
11      printf("fd2 = %d\n", fd2);
12      exit(0);
13  }
```

11.7 ◆

修改图 11.4 中所示的 cpfile 程序，使得它用 Rio 函数从标准输入拷贝到标准输出，一次 MAXBUF 个字节。

11.8 ◆◆

编写图 11.10 中的 statcheck 程序的一个版本，叫做 fstatcheck，它从命令行上取得一个描述符数字而不是文件名。

11.9 ◆◆

考虑下面对习题 11.8 中的对 fstatcheck 程序的调用：

```
unix> fstatcheck 3 < foo.txt
```

你可能会预想这个对 fstatcheck 的调用将提取和显示文件 foo.txt 的元数据。然而，当我们在我们的系统上运行它时，它将失败，返回“坏的文件描述符”。根据这种情况，填写 shell 在 fork 和 execve 调用之间必须执行的伪代码：

```
if (Fork() == 0) { /* child */
    /* What code is the shell executing right here? */
    Execve("fstatcheck", argv, envp);
}
```

11.10 ◆◆

修改图 11.4 中的 cpfile 程序，使得它有一个可选的命令行参数 infile。如果给定了 infile，那么拷

贝 infile 到标准输出，否则像以前那样拷贝标准输入到标准输出。一个要求是对于两种情况，你的解答都必须使用原来的拷贝循环（第 9~11 行）。只允许你插入代码，而不允许更改任何已存在的代码。

练习题答案

练习题 11.1 答案

Unix 进程生命周期开始时，打开的描述符赋给了 stdin（描述符 0）、stdout（描述符 1）和 stderr（描述符 2）。open 函数总是返回最低的未打开的描述符，所以第一次调用 open 会返回描述符 3。调用 close 函数会释放描述符 3。最后对 open 的调用会返回描述符 3，因此程序的输出是“fd2=3”。

练习题 11.2 答案

描述符 fd1 和 fd2 每个都有各自的打开文件表表项，所以每个描述符对于 foobar.txt 都有它自己的文件位置。因此，从 fd2 的读操作会读取 foobar.txt 的第一个字节，并输出

```
c = f
```

而不是像你开始可能想的

```
c = o
```

练习题 11.3 答案

回想一下，子进程会继承父进程的描述符表，以及所有进程共享的同一个打开文件表。因此，描述符 fd 在父子进程中都指向同一个打开文件表表项。当子进程读取文件的第一个字节时，文件位置加 1。因此，父进程会读取第二个字节，而输出就是

```
c = o
```

练习题 11.4 答案

重定向标准输入（描述符 0）到描述符 5，我们将调用 dup2(5,0) 或者等价的 dup2(5,STDIN_FILENO)。

练习题 11.5 答案

第一眼你会想输出应该是

```
c = f
```

但是因为我们将 fd1 重定向到了 fd2，输出实际上是

```
c = o
```

CHAPTER 12

网络编程

网络应用随处可见。任何时候你浏览Web、发送email信息或者弹出一个X window，你就正在使用一个网络应用程序。有趣的是，所有的网络应用都是基于相同的基本编程模型，有着相似的整体逻辑结构，并且依赖相同的编程接口。

网络应用依赖于很多在我们的系统研究中已经学习过的概念。例如，进程、信号、字节排序、存储器（memory）映射以及动态存储分配，都扮演着重要的角色。即使是对于专家而言，也还是有一些新的概念。我们需要理解基本的客户端-服务器编程模型，以及如何编写使用因特网提供的服务的客户端-服务器程序。最后，我们将把所有这些概念结合起来，开发一个小但是功能齐全的Web服务器，能够为真实的Web浏览器提供静态和动态的文本和图形内容。

12.1 客户端-服务器编程模型

每个网络应用都是基于客户端-服务器模型的。根据这个模型，一个应用是由一个服务器进程和一个或者多个客户端进程组成。服务器管理某种资源，并且通过操作这种资源来为它的客户端提供某种服务。例如，一个Web服务器管理了一组磁盘文件，它会代表客户端进行检索和执行。一个FTP服务器就管理了一组磁盘文件，它会为客户端进行存储和检索。相似地，一个电子邮件服务器管理了一些文件，它为客户端进行读和更新。

客户端-服务器模型中的基本操作是事务（transaction）（图12.1）。一个客户端-服务器事务由四步组成：

1. 当一个客户端需要服务时，它向服务器发送一个请求，发起一个事务。例如，当Web浏览器需要一个文件时，它就发送一个请求给Web服务器。

2. 服务器收到请求后，解释它，并以适当的方式操作它的资源。例如，当Web服务器收到浏览器发出的请求后，它就读一个磁盘文件。

3. 服务器给客户端发送一个响应，并等待下一个请求。例如，Web服务器将文件发送回客户端。

4. 客户端收到响应并处理它。例如，当Web浏览器收到来自服务器的一页后，它就在屏幕上显示此页。

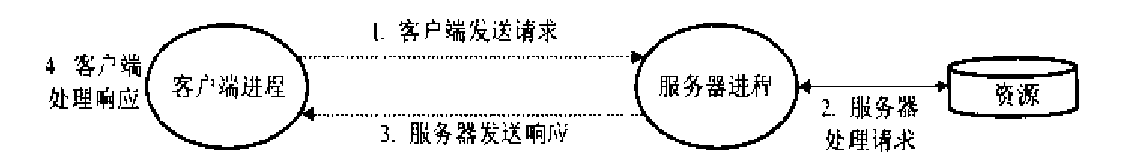


图12.1 一个客户端-服务器事务

认识到客户端和服务器是进程，而不是在本上下文中常被称为的机器或者主机，这是很重要的。一台主机可以同时运行许多不同的客户端和服务器，而且客户端和服务器的事务可以在同一台或是不同的主机上。无论客户端和服务器是怎样映射到主机上的，客户端-服务器模型是相同的。

旁注：客户端-服务器事务与数据库事务

客户端-服务器事务不是数据库事务，而且也没有数据库事务的特性，例如原子性。在我们的上下文中，事务仅仅是客户端和服务器之间执行的一系列步骤。

12.2 网络

客户端和服务器通常运行在不同的主机上，并且通过计算机网络的硬件和软件资源来通信。网络是复杂的系统，在这里我们只想了解一点皮毛。我们的目标是从程序员的角度给你一个可工作的思考模型。

对于一个主机而言，网络只是又一种I/O设备，作为数据源和数据接收方，如图12.2所示。一个插到I/O总线扩展槽的适配器提供了到网络的物理接口。从网络上接收到的数据从适配器经过I/O和存储器总线拷贝到存储器，典型地是通过DMA（直接存储器存取方式）传送。相似地，数据也能从存储器拷贝到网络。

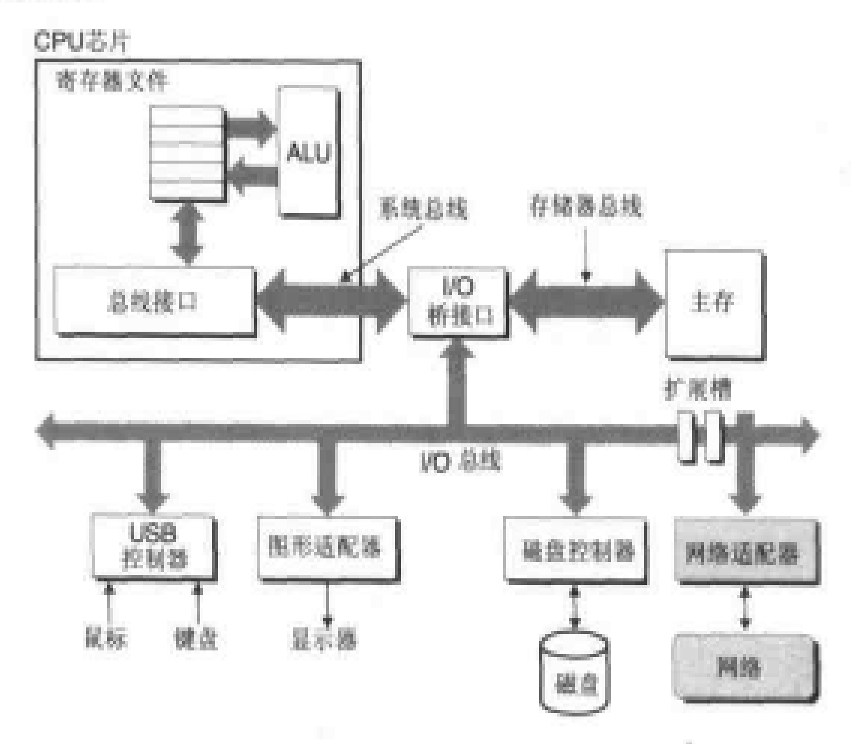


图12.2　一个网络主机的硬件组成

物理上而言，网络是一个按照地理远近组成的层次系统。最低层是LAN（Local Area Network，局域网），范围在一个建筑或者校园内。迄今为止，最流行的局域网技术是以太网（Ethernet），它是由施乐公司帕洛阿尔托研究中心（Xerox PARC）在20世纪70年代中期提出的。以太网技术被证明在3Mb/s～1Gb/s之间都是相当适合的。

一个以太网段（Ethernet segment）包括一些电缆（通常是双绞线）和一个叫做集线器的小盒子，如图12.3所示。以太网段通常服务于一个小的区域，例如某建筑物的一个房间或者一个楼层。每根电缆都有相同的最大位带宽，典型的是100Mb/s或者1Gb/s。一端连接到主机的适配器，而另一端则连接到集线器的一个端口上。集线器不加分辨地将从一个端口上收到的每个位复制到其他所有的端口上。因此，每台主机都能看到每个位。

每个以太网适配器都有一个全球惟一的48位地址，它存储在这个适配器的永久性存储器上。一台主机可以发送一段位，称为帧（frame），到这个网段内其他任何主机。每个帧包括一些固定数量的头（header）位，用来标识此帧的源和目的地址以及此帧的长度，此后紧随的就是数据位的有效

载荷（payload）。每个主机适配器都能看到这个帧，但是只有目的主机实际读取它。

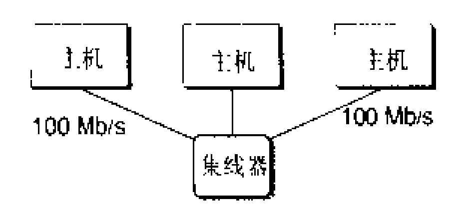


图 12.3 以太网分段

使用一些电缆和叫做网桥（bridge）的小盒子，多个以太网段可以连接成较大的局域网，称为桥接以太网（bridged Ethernet），如图 12.4 所示。桥接以太网能够跨越整个建筑物或者校区。在一个桥接以太网里，一些电缆连接网桥与网桥，而另外一些连接网桥和集线器。这些电缆的带宽可以是不同的。在我们的示例中，网桥与网桥之间的电缆有 1Gb/s 的带宽，而四根网桥和集线器之间电缆的带宽却是 100Mb/s。

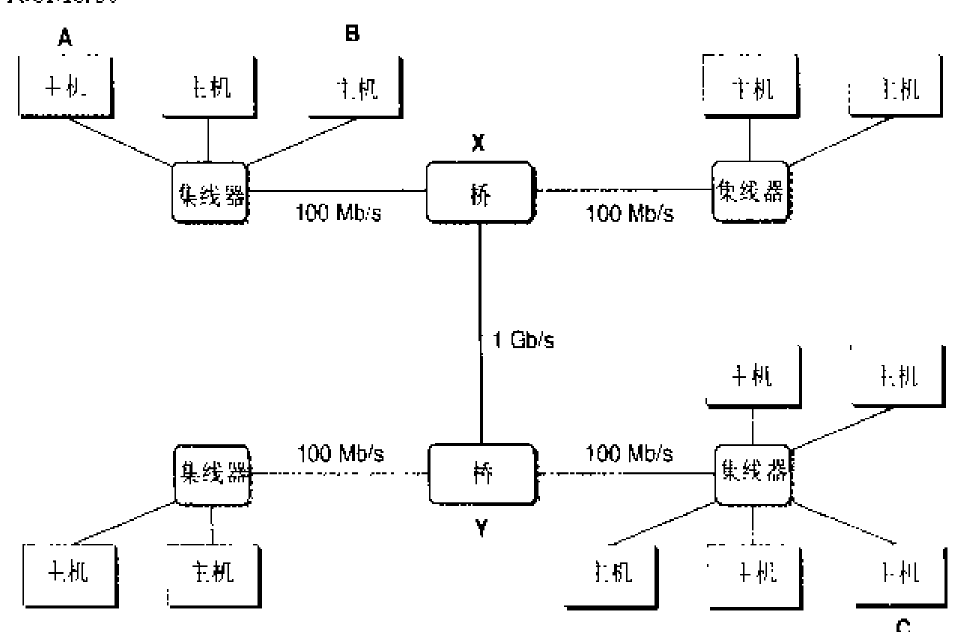


图 12.4 桥接以太网

网桥比集线器更充分地利用了电缆带宽。利用一种聪明的分配算法，它们随着时间自动学习哪个主机可以通过哪个端口可达，然后在有必要时，有选择地将帧从一个端口拷贝到其他端口。例如，如果主机 A 发送一个帧到同网段上的主机 B，当该帧到达网桥 X 的输入端口时，它将丢弃此帧，因而节省了其他网段上的带宽。然而，如果主机 A 发送一个帧到一个不同网段上的主机 C，那么网桥 X 只会把此帧拷贝到和网桥 Y 相连的端口上，网桥 Y 会只把此帧拷贝到与主机 C[1] 的网段连接的端口。

为了简化局域网的表示，我们将把集线器和网桥以及连接它们的电缆画成一根水平线，如图 12.5 所示。

在层次的更高级别中，多个不兼容的局域网可以通过叫做路由器（router）的特殊计算机连接起来，组成一个 internet（互联网络）。

1 此处原文为网桥 C bridge C's，但我认为应该是主机 C，host C's。——译者

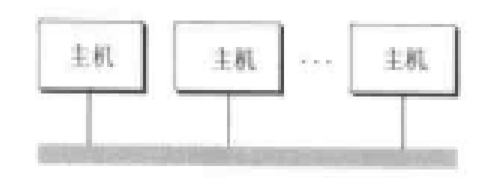


图 12.5 局域网的概念视图

旁注：Internet 和 internet

我们经常用小写字母的 internet 描述一般概念，而用大写字母的 Internet 来描述一种特殊的实际应用，也就是所谓的全球 IP 因特网。

每台路由器对于它所连接的每个网络都有一个适配器（端口）。路由器也能连接高速点到点电话连接，这是 WAN（Wide-Area Network，广域网）的一种示例，之所以这么叫是因为它们覆盖的地理范围比局域网的大。一般而言，路由器可以用来由各种局域网和广域网构建 internet（互联网络）。例如，图 12.6 展示了一个 internet 示例，3 台路由器连接了一对局域网和广域网。

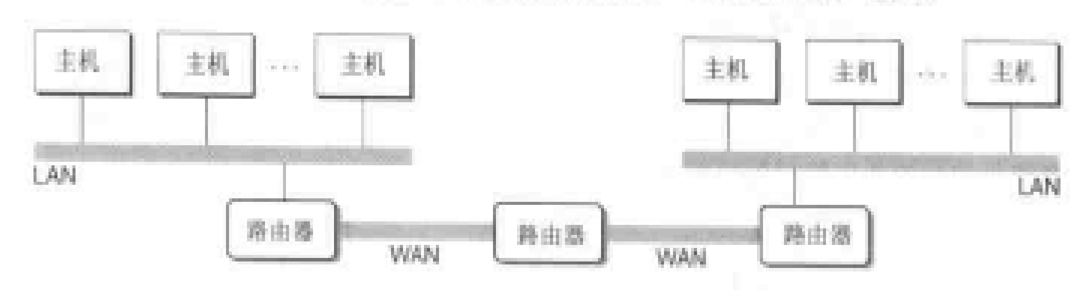


图 12.6 一个小型的 internet（互联网络）

两个局域网和两个广域网用三台路由器连接起来。

internet（互联网络）至关重要的特性是，它能由采用完全不同和不兼容技术的各种局域网和广域网组成。每台主机和其他每台主机都是物理相连的，但是如何使得某台*源主机*跨过所有这些不兼容的网络发送数据位到另一台*目的主机*成为可能呢？

解决办法是一层运行在每台主机和路由器上的*协议软件*，它消除了不同网络之间的差异。这个软件执行一种协议，控制主机和路由器如何协同工作来实现数据传输。这种协议必需提供两种基本能力：

- *命名方法*。不同的局域网技术有不同和不兼容的方式来为主机分配地址。internet（互联网络）协议通过定义一种一致的主机地址格式，消除了这些差异。每台主机会被分配至少一个这种 *internet 地址*（internet address），这个地址惟一地标识了它。
- *传送机制*。在电缆上编码位和将这些位封装成帧方面，不同的网络互联技术有不同的和不兼容的方式。internet（互联网络）协议通过定义一种把数据位捆扎成不连续的组块（chunk）——也就是包——的统一方式，从而消除了这些差异。一个包是由*包头*（header）和*有效载荷*（payload）组成的，其中包头括包的大小以及源主机和目的主机的地址，有效载荷包括从源主机发出的数据位。

图 12.7 展示了一个主机和路由器如何使用 internet（互联网络）协议在不兼容的局域网间传送数据的示例。这个 internet（互联网络）示例由两个局域网通过一台路由器连接而成。一个客户端运

行在主机 A 上，主机 A 与 LAN1 相连，它发送了一串数据字节到运行在主机 B 上的服务器端，主机 B 则连接在 LAN2 上。这个过程有 8 个基本步骤：

1. 运行在主机 A 上的客户端进行了一个系统调用，从客户端的虚拟地址空间拷贝数据到内核缓冲区。

2. 主机 A 上的协议软件通过在数据前附加 internet（互联网络）包头和 LAN1 帧头，创建了一个 LAN1 的帧。internet（互联网络）包头寻址到 internet（互联网络）主机 B。LAN1 帧头寻址到路由器。然后它传送此帧到适配器。注意，LAN1 帧的有效载荷是一个 internet（互联网络）包，其有效载荷是实际的用户数据。这种封装是基本的网络互联方法之一。

3. LAN1 适配器拷贝该帧到网络上。

4. 当此帧到达路由器时，路由器的 LAN1 适配器从电缆上读取它，并把它传送到协议软件。

5. 路由器从 internet 包头中提取出目的 internet 地址，并用它作为路由表的索引，确定向哪里转发这个包，在本例中是 LAN2。路由器剥落旧的 LAN1 的帧头，加上寻址到主机 B 的新的 LAN2 帧头，并把得到的帧传送到适配器。

6. 路由器的 LAN2 适配器拷贝该帧到网络上。

7. 当此帧到达主机 B 时，它的适配器从电缆上读到此帧，并将它传送到协议软件。

8. 最后，主机 B 上的协议软件剥落包头和帧头。当服务器进行一个读取这些数据的系统调用时，协议软件最终将得到的数据拷贝到服务器的虚拟地址空间。

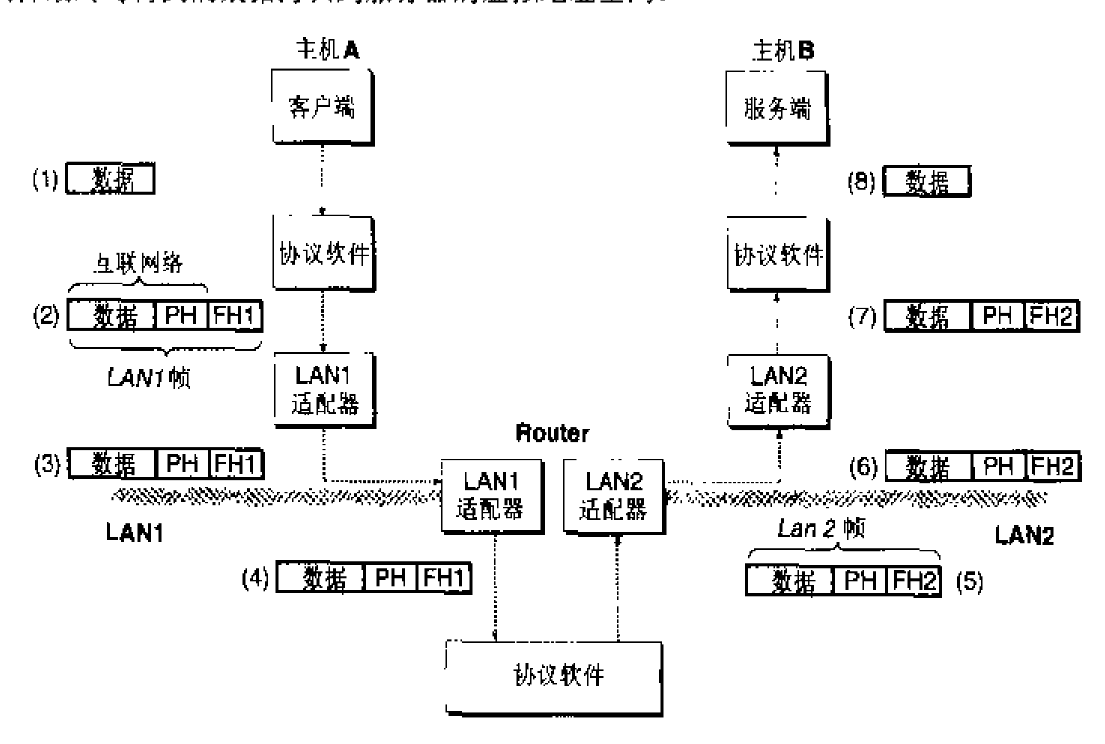


图 12.7 在 internet（互联网络）上，数据是如何从一台主机传送到另一台主机的

关键词：PH：internet（互联网络）包头；FH1：LAN1 的帧头；FH2：LAN2 的帧头。

当然，在这里我们掩盖了许多复杂的问题。如果不同的网络有不同帧大小的最大值，该怎么办呢？路由器如何知道往哪里转发帧呢？当网络拓扑变化时，如何通知路由器？如果一个包丢失了又会如何呢？虽然如此，我们的示例抓住了 internet（互联网络）思想的精髓，封装是关键。

12.3 全球 IP 因特网

全球 IP 因特网是 internet（互联网络）最著名和最成功的实现。从 1969 年起，它就以这样或那样的形式存在了。虽然因特网的内部体系结构复杂而且不断变化，但是自从 20 世纪 80 年代早期以来，客户端-服务器应用的组织就一直保持相当的稳定。图 12.8 展示了一个因特网客户端-服务器应用程序的基本硬件和软件组织。

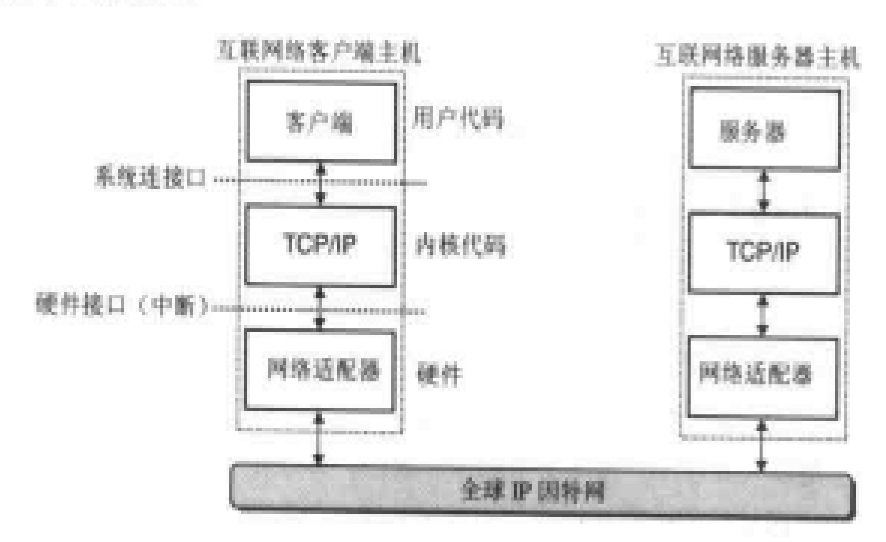


图 12.8 一个因特网应用程序的硬件和软件组织

每台因特网主机都运行实现 TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/互联网络协议）的软件，几乎每个现代计算机系统都支持这个协议。因特网的客户端和服务器混合使用**套接字接口**函数和 Unix I/O 函数来进行通信（我们将在 12.4 节中介绍套接字接口）。套接字函数典型地是作为系统调用来实现的，这些系统会陷入内核，并调用各种内核模式的 TCP/IP 函数。

TCP/IP 实际是一个协议族，其中每一个都提供不同的功能。例如，IP 协议提供基本的命名方法和递送机制，这种递送机制能够从一台因特网主机往其他主机发送包，也叫做**数据报**（datagram）。IP 机制从某种意义上而言是不可靠的，因为，如果数据报在网络中丢失或者重复，它并不会恢复。UDP（不可靠数据报协议）稍微扩展了 IP 协议，这样一来，包可以在进程间而不是在主机间传送。TCP 是一个建筑在 IP 之上的复杂协议，提供了进程间可靠的全双工（双向的）连接。为了简化我们的讨论，我们将 TCP/IP 看做是一个单独的整体协议。我们将不讨论它的内部工作，只讨论 TCP 和 IP 为应用程序提供的某些基本功能。我们将不讨论 UDP。

从程序员的角度，我们可以把因特网看做一个世界范围的主机集合，满足以下特性：

- 主机集合被映射为一组 32 位的 **IP 地址**。
- 这组 IP 地址被映射为一组称为**因特网域名**（Internet domain name）的标识。
- 一个因特网主机上的进程能够通过一个**连接**（connection）和任何其他因特网主机上的进程通信。

下三节将更详细地讨论这些基本的因特网概念。

12.3.1 IP地址

一个IP地址就是一个32位无符号整数。网络程序将IP地址存放在如图12.9所示的IP地址结构中。

netinet/in.h

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big endian) */
};
```

netinet/in.h

图12.9 IP地址结构

旁注：为什么要用结构来存放标量IP地址？

把一个标量地址存放在结构中，是套接字接口早期实现的不幸产物。为IP地址定义一个标量类型该更有意义，但是现在更改已经太迟了，因为已经有大量应用是基于此的。

因为因特网主机可以有不同的主机字节顺序，TCP/IP为任意整数数据项定义了一致的网络字节排序（network byte order）（大端字节顺序），例如IP地址，它放在包头中，通过网络。在IP地址结构中存放的地址总是以（大端法）网络字节顺序存放的，即使主机字节顺序（host byte order）是小端法。Unix提供了下面这样的函数在网络和主机字节顺序间实现转换：

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
                                        返回：按照网络字节顺序的值。
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
                                        返回：按照主机字节顺序的值。
```

hotnl函数将32位整数由主机字节顺序转换为网络字节顺序。ntohl函数将32位整数从网络字节顺序转换为主机字节。htons和ntohs函数为16位的整数执行相应的转换。

IP地址是以点分十进制表示法表示，这里，每个字节由它的十进制值表示，并且用句点和其他字节间分开。例如，128.2.194.242就是地址0x8002c2f2的点分十进制表示。在Linux系统上，你能够使用HOSTNAME命令来设置你自己主机的点分十进制地址：

```
linux> hostname -i
128.2.194.242
```

因特网程序使用inet_aton和inet_ntoa函数来实现IP地址和点分十进制串之间的转换：

```
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
                                        返回：若成功则为1，若出错则为0。
char *inet_ntoa(struct in_addr in);
                                        返回：指向点分十进制串的指针。
```

inet_aton 函数将一个点分十进制串（cp）转换为一个网络字节顺序的 IP 地址（inp）。相似地，inet_ntoa 函数将一个网络字节顺序的 IP 地址转换为它所对应的点分十进制串。注意，对 inet_aton 的调用传递的是指向结构的指针，而对 inet_ntoa 的调用传递的是结构本身。

旁注：ntoa 和 aton 是什么意思？

"n" 表示的是网络（network），"a" 表示应用（application），而 "to" 表示转换。

练习题 12.1

完成下表：

十六进制地址	点分十进制地址
0x0	
0xffffffff	
0xef000001	
	205.188.160.121
	64.12.149.13
	205.188.146.23

练习题 12.2

编写程序 hex2dd.c，它将它的十六进制参数转换为点分十进制串并打印出结果。例如

```
unix> ./hex2dd 0x8002c2f2
128.2.194.242
```

练习题 12.3

编写程序 dd2hex.c，它将它的点分十进制参数转换为十六进制数并打印出结果。例如

```
unix> ./dd2hex 128.2.194.242
0x8002c2f2
```

12.3.2 因特网域名

因特网客户端和服务器互相通信时使用的是 IP 地址。然而，对于人们而言，大整数是很难记住的，所以因特网也定义了一组更加人性化的域名（domain name），以及一种将域名映射到 IP 地址的机制。域名是一串用句点分隔的单词（字母、数字和破折号），例如

```
kittyhawk.cmcl.cs.cmu.edu
```

域名集合形成了一个层次结构，每个域名编码了它在这个层次中的位置。通过一个示例你将很容易理解这点。图 12.10 展示了域名层次结构的一部分。层次结构被表示为一棵树。树的节点表示域名，反向到根的路径形成了域名。子树称为子域（subdomain）。层次结构中的第一层是一个未命名的根节点。下一层是一组第一层域名（first-level domain names），由非赢利组织 ICANN（Internet Corporation for Assigned Names and Numbers，因特网分配名字数字协会）定义。常见的第一层域名包括 com、edu、gov、org 和 net。

下一层是第二层（second-level）域名，例如 cmu.edu，这些域名是由 ICANN 的各个授权代理

按照先到先服务的基础分配的。一旦一个组织得到了一个第二层域名，那么它就可以在这个子域中创建任何新的域名了。

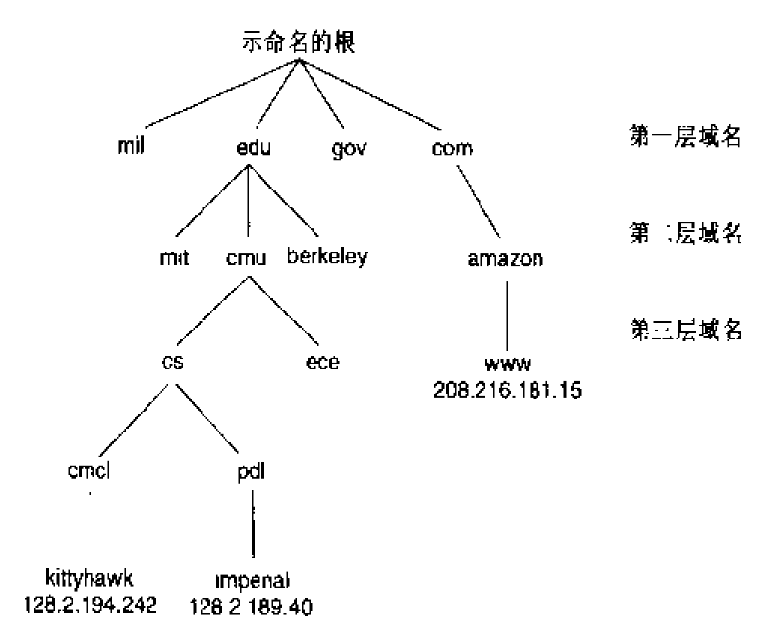


图 12.10 因特网域名层次结构的一部分

因特网定义了域名集合和 IP 地址集合之间的映射。直到 1988 年，这个映射都是通过一个叫做 HOSTS.TXT 的文本文件来手工维护的。从那以后，这个映射是通过分布世界范围内的数据库——称为 DNS（域名系统）——来维护的。从概念上而言，DNS 数据库由上百万的图 12.11 所示的主机条目结构（host entry structure）组成的，其中每条定义了一组域名（一个官方名字和一组别名）和一组 IP 地址之间的映射。从数学意义上讲，你可以认为，每条主机条目就是一个域名和 IP 地址的等价类。

netdb.h

```
/* DNS host entry structure */
struct hostent {
    char *h_name;         /* official domain name of host */
    char **h_aliases;     /* null-terminated array of domain names */
    int h_addrtype;       /* host address type (AF_INET) */
    int h_length;         /* length of an address, in bytes */
    char **h_addr_list;   /* null-terminated array of in_addr structs */
};
```

netdb.h

图 12.11 DNS 主机条目结构

因特网应用程序通过调用 gethostbyname 和 gethostbyaddr 函数，从 DNS 数据库中检索任意的主机条目。

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

返回：若成功则为非 NULL 指针，若出错则为 NULL 指针，同时设置 h_errno.

```
struct hostent *gethostbyaddr(const char *addr, int len, 0);
```

返回：若成功则为非 NULL，若出错则为 NULL 指针，同时设置 h_errno.

gethostbyname 函数返回和域名 name 相关的主机条目。gethostbyaddr 函数返回和 IP 地址 addr 相关的主机条目。第二个参数给出了一个 IP 地址的字节长度，对于目前的因特网而言总是四个字节。对于我们的要求来说，第三个参数总是零。

我们可以借助于图 12.12 中的 HOSTINFO 程序，来挖掘一些 DNS 映射的特性，这个程序从命令行读取一个域名或点分十进制地址，并显示相应的主机条目。

code/netp/hostinfo.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       char **pp;
6       struct in_addr addr;
7       struct hostent *hostp;
8
9       if (argc != 2) {
10          fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
11                  argv[0]);
12          exit(0);
13      }
14
15      if (inet_aton(argv[1], &addr) != 0)
16          hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
17      else
18          hostp = Gethostbyname(argv[1]);
19
20      printf("official hostname: %s\n", hostp->h_name);
21
22      for (pp = hostp->h_aliases; *pp != NULL; pp++)
23          printf("alias: %s\n", *pp);
24
25      for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
26          addr.s_addr = *((unsigned int *)*pp);
27          printf("address: %s\n", inet_ntoa(addr));
28      }
29      exit(0);
30  }
```

code/netp/hostinfo.c

图 12.12 检索并打印 DNS 主机条目

每台因特网主机都有本地定义的域名 localhost，这个域名总是映射为本地回送地址（loopback address）127.0.0.1：

```
unix> ./hostinfo localhost
official hostname: localhost
```

```
alias: localhost.localdomain
address: 127.0.0.1
```

localhost 名字为引用运行在同一台机器上的客户端和服务器提供了一种便利和可移植的方式，这对调试相当有用。我们可以使用 HOSTNAME 来确定我们本地主机的实际域名：

```
unix> ./hostname
kittyhawk.cmcl.cs.cmu.edu
```

在最简单的情况中，一个域名和一个 IP 地址之间是一一映射：

```
unix> ./hostinfo kittyhawk.cmcl.cs.cmu.edu
official hostname: kittyhawk.cmcl.cs.cmu.edu
address: 128.2.194.242
```

然而，在某些情况下，多个域名可以映射为同一个 IP 地址：

```
unix> ./hostinfo cs.mit.edu
official hostname: EECS.MIT.EDU
alias: cs.mit.edu
address: 18.62.1.6
```

在最通常的情况下，多个域名可以映射到多个 IP 地址：

```
unix> ./hostinfo www.aol.com
official hostname: aol.com
alias: www.aol.com
address: 205.188.160.121
address: 64.12.149.13
address: 205.188.146.23
```

最后，我们注意到某些合法的域名没有映射到任何 IP 地址：

```
unix> ./hostinfo edu
Gethostbyname error: No address associated with name
unix> ./hostinfo cmcl.cs.cmu.edu
Gethostbyname error: No address associated with name
```

旁注：有多少因特网主机？

因特网软件协会（Internet Software Consortium，www.isc.org）自从 1987 年以后，每年进行两次因特网域名调查。这个调查，通过计算已经分配给一个域名的 IP 地址的数量来估算因特网主机的数量，展示了一种令人吃惊的趋势。自从 1987 年以来，当时一共大约有 20 000 台因特网主机，每年主机数量都大概会翻一番。到 2001 年 6 月，全球已经有超过 120 000 000 台因特网主机了。

练习题 12.4

编译图 12.12 中的 HOSTINFO 程序，然后在你的系统上连续运行 hostinfo.aol.com 三次。

A. 在三个主机条目的 IP 地址顺序中，你注意到了什么？

B. 这种顺序有何作用？

12.3.3 因特网连接

因特网客户端和服务器通过在连接（connection）上发送和接收字节流来通信。从连接一对进程的意义上而言，连接是点对点（point-to-point）的。从数据可以同时双向流动的角度来说，它是全双工（full-duplex）的。并且从——除了一些如粗心的耕锄机操作员切断了电缆引起灾难性的失败以外——由源进程发出的字节流最终被目的进程以它发出的顺序收到它的角度来说，它也是可靠的。

套接字（socket）是连接的端点（end-point）。每个套接字都有相应的套接字地址，是由一个因特网地址和一个16位的整数端口组成的，用“地址：端口”来表示。当客户端发起一个连接请求时，客户端套接字地址中的端口是由内核自动分配的，称为临时端口（ephemeral port）。然而，服务器套接字地址中的端口通常是某个知名的端口，是和服务对应的。例如，Web 服务器通常使用端口 80，而电子邮件服务器使用端口 25。在 Unix 机器上，文件/etc/services 包含一张这台机器提供的服务以及它们的知名端口号的综合列表。

一个连接是由它两端的套接字地址惟一确定的。这对套接字地址叫做套接字对（socket pair），由下列三元组来表示的：

```
(cliaddr:clipart, servaddr:servport)
```

其中 cliaddr 是客户端的 IP 地址，cliport 是客户端的端口，servaddr 是服务器的 IP 地址，而 servport 是服务器的端口。例如，图 12.13 展示了一个 Web 客户端和一个 Web 服务器之间的连接。在这个示例中，Web 客户端的套接字地址是

```
128.2.194.242:51213
```

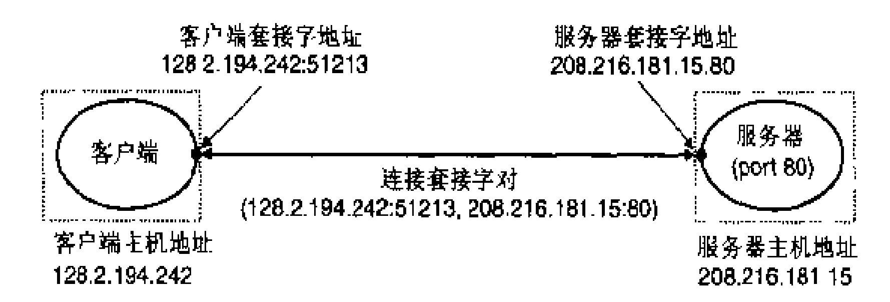


图 12.13 因特网连接的分析

其中端口号 51213 是内核分配的临时端口号。Web 服务器的套接字地址是：

```
208.216.181.15:80
```

其中端口号 80 是和 Web 服务相关联的知名端口号，给定这些客户端和服务器套接字地址，客户端和服务器之间的连接就由下列套接字对惟一确定了：

```
(128.2.194.242:51213, 208.216.181.15:80)
```

旁注：因特网的起源

因特网是政府、学校和工业界合作的最成功的示例之一。它成功的因素很多，但是我们认为有两点尤其重要：美国政府 30 年持续不变的投资，以及充满激情的研究人员对麻省理工大学的 Dave Clarke 提出的“粗略一致和能用的代码”的投入。

因特网的种子是在 1957 年播下的，其时，正值冷战的高峰，苏联发射 Sputnik，第一颗人造地球卫星，震惊了世界。作为响应，美国政府创建了高级研究计划署（ARPA），其任务就是重建美国

在科学与技术上的领导地位。1967年，ARPA的Lawrence Roberts提出了一个计划，建立一个叫做ARPANET的新网络。第一个ARPANET节点是在1969年建立并运行的。到1971年，有了13个ARPANET节点，而且email作为第一个重要的网络应用涌现出来。

1972年，Robert Kahn概括了网络互联的一般原则：一组互相连接的网络，通过叫做"路由器"的黑盒子按照"尽力传送基础"在互相独立处理的网络间实现通信。1974年，Kahn和Vinton Cerf发表了TCP/IP协议的第一本详细资料，到1982年它成为了ARPANET的标准网络互联协议。1983年1月1日，ARPANET的每个节点都切换到TCP/IP，标志着全球IP因特网的诞生。

1985年，Paul Mockapetris发明了DNS，有1 000多台因特网主机。次年，国家科学基金会（NSF）用56Kb/s的电话线连接了13个节点，构建了NSFNET的骨干网。其后在1988年升级到1.5Mb/s T1的连接速率，1991年为45Mb/s T3的连接速率。到1988年，有超过50 000台主机。1989年，原始的ARPANET正式退休了。1995年，已经有几乎10 000 000台因特网主机了，NSF取消了NSFNET，并且用基于由一打左右的公众网络接入点连接的私有商业骨干网的现代因特网架构取代了它。

12.4 套接字接口

套接字接口（socket interface）是一组用来结合Unix I/O函数创建网络应用的函数。大多数现代系统上都实现它，包括所有的Unix变种、Windows和Macintosh系统。图12.14给出了一个典型的客户端-服务器事务的上下文中的套接字接口。当我们讨论各个函数时，你可以使用这张图来作为向导图。

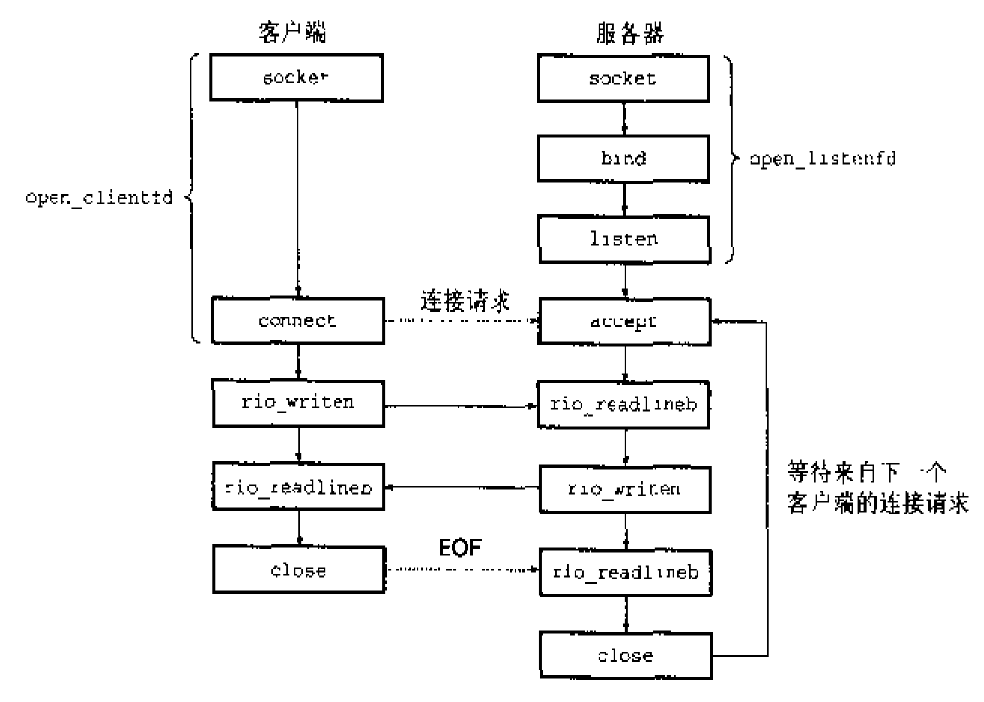


图12.14　套接字接口概述

旁注：套接字接口的起源

套接字接口是加州伯克利分校的研究人员在20世纪80年代早期提出的。因为这个原因，它也经常被叫做伯克利套接字。伯克利的研究者使得套接字接口适用于任何底层的协议。第一个实现的

就是基于 TCP/IP 协议的，他们把它包括在 Unix 4.2 BSD 的内核里，并且分发给许多学校和实验室。这在因特网的历史上是一个重大事件。几乎一夜之间，成千上万的人们接触到了 TCP/IP 和它的源代码。它引起了巨大的兴趣，并激发了新的网络和网络互联研究的浪潮。

12.4.1 套接字地址结构

从 Unix 内核的角度来看，套接字就是通信的端点（end-point）。从 Unix 程序的角度来看，套接字就是一个有相应描述符的打开文件。

因特网的套接字地址存放在如图 12.15 所示的类型为 sockaddr_in 的 16 字节结构中。对于因特网应用，sin_family 成员是 AF_INTE，sin_port 成员是一个 16 位的端口号，而 sin_addr 成员就是一个 32 位的 IP 地址。IP 地址和端口号总是以网络字节顺序（大端法）存放的。

sockaddr: socketbits.h (included by socket.h). sockaddr in: netinit/in.h

```
/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    unsigned short sa_family;    /* protocol family */
    char           sa_data[14];  /* address data. */
};

/* Internet-style socket address structure */
struct sockaddr_in {
    unsigned short sin_family;   /* address family (always AF_INET) */
    unsigned short sin_port;     /* port number in network byte order */
    struct in_addr sin_addr;     /* IP address in network byte order */
    unsigned char sin_zero[8];   /* pad to sizeof(struct sockaddr) */
};
```

sockaddr: socketbits.h (included by socket.h). sockaddr_in: netinit/in.h

图 12.15 套接字地址结构

in_addr 结构如图 12.9 所示。

旁注：_in 后缀意味什么？

_in 后缀是互联网络（internet）的缩写，而不是输入（input）的缩写。

connect、bind 和 accept 函数要求一个指向与协议相关的套接字地址结构的指针。套接字接口的设计者面临的问题是，如何定义这些函数，使之能接受各种类型的套接字地址结构。今天，我们可以使用通用的 void*指针，那时在 C 中并不存在这种类型的指针。解决办法是定义套接字函数要求一个指向通用 sockaddr 结构的指针，然后要求应用程序将与协议特定的结构的指针强制转换成这个通用结构。为了简化我们的代码示例，我们跟随 Steven 的指导，定义下面的类型：

```
typedef struct sockaddr SA;
```

然后无论何时我们需要将 sockaddr_in 结构强制转换成通用 sockaddr 结构时，我们都使用这个类型（参见图 12.16 的第 20 行的示例）。

12.4.2 socket 函数

客户端和服务器使用 socket 函数来创建一个套接字描述符（socket descriptor）。

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

返回：若成功则为非负描述符，若出错则为-1。

在我们的代码中，我们总是带这样的参数来调用socket函数：

```
clientfd=Socket(AF_INET, SOCK_STREAM, 0);
```

其中，AF_INET表明我们正在使用因特网，而SOCK_STREAM表示套接字是因特网连接的端点（end-point）。socket返回的clientfd描述符仅是部分打开，并且不能用于读写。我们如何完成打开套接字的工作，取决于我们是客户端还是服务器。下一节描述当我们是客户端时如何完成打开套接字的工作。

12.4.3 connect函数

客户端是通过调用connect函数来建立和服务器的连接的。

```
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen );
```

返回：若成功则为0，若出错则为-1。

connect函数试图与套接字地址为serv_addr的服务器建立一个因特网连接，其中addrlen是sizeof(sockaddr_in)。connect函数会阻塞，一直到连接成功建立或是发生错误，如果成功，sockfd描述符现在就准备好读写了，并且，得到的连接是由套接字对

```
(x:y, serv_addr.sin_addr:serv_addr.sin_port)
```

刻画的，其中x表示客户端的IP地址，而y表示临时端口，它惟一地确定了客户端主机上的客户端进程。

12.4.4 open_clientfd函数

我们发现将socket和connect函数包装成一个叫做open_clientfd的辅助函数是很方便的，客户端可以用它来和服务器建立连接。

```
#include "csapp.h"

int open_clientfd(char *hostname, int port);
```

返回：若成功则为描述符，若Unix出错则为-1，若DNS出错则为-2。

open_clientfd函数和服务器建立一个连接，该服务器运行在主机hostname上，并在知名端口port上监听连接请求。它返回一个打开的套接字描述符，该描述符准备好了，可以用Unix I/O函数做输入和输出。图12.16给出了open_clientfd的代码。

code/src/csapp.c

```
1   int open_clientfd(char *hostname, int port)
2   {
```

```
3        int clientfd;
4        struct hostent *hp;
5        struct sockaddr_in serveraddr;
6
7        if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8            return -1; /* check errno for cause of error */
9
10       /* Fill in the server's IP address and port */
11       if ((hp = gethostbyname(hostname)) == NULL)
12           return -2; /* check h_errno for cause of error */
13       bzero((char *) &serveraddr, sizeof(serveraddr));
14       serveraddr.sin_family = AF_INET;
15       bcopy((char *)hp->h_addr,
16             (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
17       serveraddr.sin_port = htons(port);
18
19       /* Establish a connection with the server */
20       if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
21           return -1;
22       return clientfd;
23   }
```

code/src/csapp.c

图 12.16 open_clientfd：和服务器建立连接的辅助函数

在创建了套接字描述符（第 7 行）后，我们为服务器检索 DNS 主机条目，并拷贝主机条目中的第一个 IP 地址（已经是按照网络字节顺序的了）到服务器的套接字地址结构（第 11～16 行）。在用按照网络字节顺序的服务器的知名端口号初始化套接字地址结构（第 17 行）之后，我们发起了一个到服务器的连接请求（第 20 行）。当 connect 函数返回时，我们返回套接字描述符给客户端，客户端就可以立即开始用 Unix I/O 和服务器通信了。

12.4.5 bind 函数

剩下的套接字函数——bind、listen 和 accept 被服务器用来和客户端建立连接。

```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

返回：若成功则为 0，若出错则为 -1。

bind 函数告诉内核将 my_addr 中的服务器套接字地址和套接字描述符 sockfd 联系起来。参数 addrlen 就是 sizeof(sockaddr_in)。

12.4.6 listen 函数

客户端是发起连接请求的主动实体。服务器是等待来自客户端的连接请求的被动实体。默认情况下，内核会认为 socket 函数创建的描述符对应于主动套接字（active socket），它存在于一个连接的客户端。服务器调用 listen 函数告诉内核，描述符是被服务器而不是客户端使用的。

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

返回：若成功则为 0，若出错则为-1。

listen 函数将 sockfd 从一个主动套接字转化为一个监听套接字（listening socket），该套接字可以接受来自客户端的连接请求。backlog 参数暗示了内核在开始拒绝连接请求之前，应该放入队列中等待的未完成连接请求的数量。backlog 参数的确切含义要求对 TCP/IP 协议的理解，这超出了我们讨论的范围。通常我们会把它设置为一个较大的值，比如 1024。

12.4.7 open_listenfd 函数

我们发现将 socket、bind 和 listen 函数结合成一个叫做 open_listenfd 的辅助函数是很有帮助的，服务器可以用它来创建一个监听描述符。

```
#include "csapp.h"

int open_listenfd(int port);
```

返回：若成功则为描述符，若 Unix 出错则为-1。

open_listenfd 函数打开和返回一个监听描述符，这个描述符准备好在知名端口 port 上接收连接请求。图 12.17 展示了 open_listenfd 的代码。在我们创建了 listenfd 套接字描述符之后，我们使用 setsockopt 函数（在这里没有描述）来配置服务器，使得它能被立即终止和重启。默认时，一个重启的服务器将在大约 30 秒内拒绝客户端的连接请求，严重地阻碍了调试。

code/src/csapp.c

```
1   int open_listenfd(int port)
2   {
3       int listenfd, optval=1;
4       struct sockaddr_in serveraddr;
5
6       /* Create a socket descriptor */
7       if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
8           return -1;
9
10      /* Eliminates "Address already in use" error from bind. */
11      if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
12                     (const void *)&optval , sizeof(int)) < 0)
13          return -1;
14
15      /* Listenfd will be an endpoint for all requests to port
16         on any IP address for this host */
17      bzero((char *)&serveraddr, sizeof(serveraddr));
18      serveraddr.sin_family = AF_INET;
19      serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
20      serveraddr.sin_port = htons((unsigned short)port);
```

```
21        if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
22            return -1;
23
24        /* Make it a listening socket ready to accept connection requests */
25        if (listen(listenfd, LISTENQ) < 0)
26            return -1;
27        return listenfd;
28    }
```
code/src/csapp.c

图 12.17 open_listenfd：打开和返回一个监听套接字的辅助函数

接下来，我们初始化服务器的套接字地址结构，为调用 bind 函数做准备。在这个例子中，我们用 INADDR_ANY 通配符地址来告诉内核这个服务器将接受来自这台主机的任何 IP 地址（第 19 行）和到知名端口 port（第 20 行）的请求。注意，我们用 htonl 和 htons 函数将 IP 地址和端口号从主机字节顺序转换为网络字节顺序。最后，我们将 listenfd 转换为一个监听描述符（第 25 行），并将它返回给调用者。

12.4.8 accept 函数

服务器通过调用 accept 函数来等待来自客户端的连接请求：

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```
返回：若成功则为非负连接描述符，若出错则为-1。

accept 函数等待来自客户端的连接请求到达侦听描述符 listenfd，然后在 addr 中填写客户端的套接字地址，并返回一个已连接描述符（connected descriptor），这个描述符可被用来利用 Unix I/O 函数与客户端通信。

监听描述符和已连接描述符之间的区别使很多同学感到迷惑。监听描述符是作为客户端连接请求的一个端点。典型地，它被创建一次，并存在于服务器的整个生命周期。已连接描述符是客户端和服务器之间已经建立起来了的连接的一个端点。服务器每次接受连接请求时，都会创建一次，只存在于服务器为一个客户端服务的过程中。

图 12.18 描绘了监听描述符和已连接描述符的角色。在第一步中，服务器调用 accept，等待连接请求到达监听描述符，具体地我们设定为描述符 3。回忆一下，描述符 0~2 预留给了标准文件。

在第二步中，客户端调用 connect 函数，发送一个连接请求到 listenfd。第三步，accept 函数打开了一个新的已连接描述符 connfd（我们假设是描述符 4），在 clientfd 和 connfd 之间建立连接，并且随后返回 connfd 给应用程序。客户端也从 connect 返回，在这一点以后，客户端和服务器就分别可以通过读和写 clientfd 和 connfd 来回传送数据了。

旁注：为何要有监听描述符和已连接描述符之间的区别？

你可能很想知道为什么套接字接口要区别监听描述符和已连接描述符。乍一看，这像是不必要的复杂化。然而，区分这两者被证明是很有用的，因为它使得我们可以建立并发服务器，它能够同

时处理许多客户端连接。例如，每次一个连接请求到达监听描述符时，我们可以派生（fork）一个新的进程，它通过它的已连接描述符与客户端通信。你将在第13章中学习更多关于并发服务器的内容。

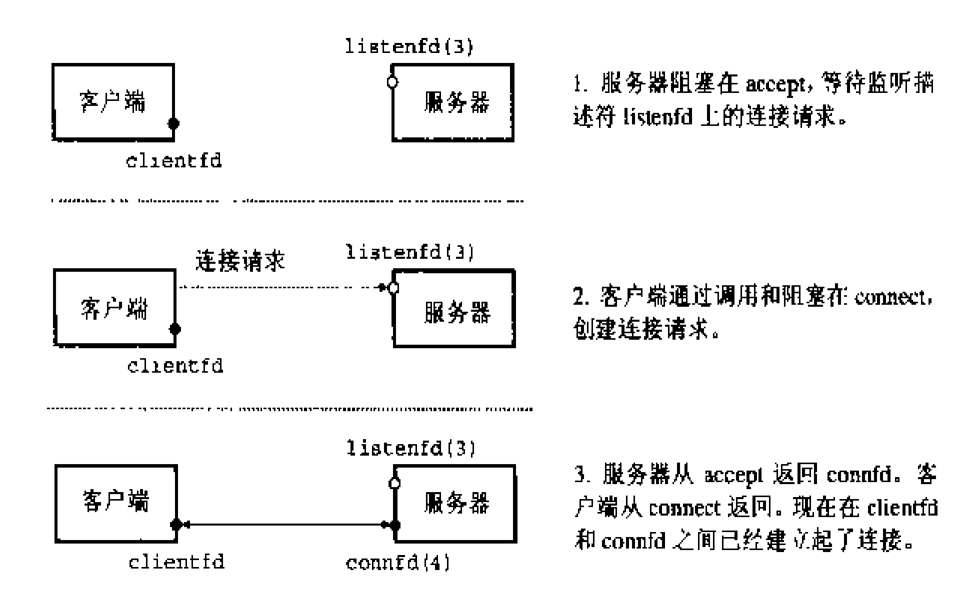


图 12.18 监听描述符和已连接描述符的角色

12.4.9 echo客户端和服务器的示例

学习套接字接口的最好方法是研究示例代码。图12.19展示了一个echo客户端的代码。在和服务器建立连接之后，客户端进入一个循环，反复从标准输入读取文本行，发送文本行给服务器，从服务器读取响应行，并输出结果到标准输出。当fgets在标准输入上遇到EOF时，或者因为用户在键盘上键入ctrl-d，或者因为在一个重定向的输入文件中用尽了所有的文本行时，循环就终止。

循环终止之后，客户端关闭描述符。这会导致发送一个EOF通知到服务器，当服务器从它的rio_readlineb函数收到一个为零的返回码时，就会检测到这个结果。在关闭它的描述符后，客户端就终止了。既然客户端内核在一个进程终止时会自动关闭所有打开的描述符，第24行的close就没有必要了。不过，显式地关闭我们已经打开的任何描述符是一个良好的编程习惯。

code/netp/echoclient.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       int clientfd, port;
6       char *host, buf[MAXLINE];
7       rio_t rio;
8
9       if (argc != 3) {
10          fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
11          exit(0);
12      }
13      host = argv[1];
14      port = atoi(argv[2]);
```

```
15
16      clientfd = Open_clientfd(host, port);
17      Rio_readinitb(&rio, clientfd);
18
19      while (Fgets(buf, MAXLINE, stdin) != NULL) {
20          Rio_writen(clientfd, buf, strlen(buf));
21          Rio_readlineb(&rio, buf, MAXLINE);
22          Fputs(buf, stdout);
23      }
24      Close(clientfd);
25      exit(0);
26  }
```
code/netp/echoclient.c

图 12.19 echo 客户端的主程序

图 12.20 展示了 echo 服务器的主程序。在打开监听描述符后，它进入一个无限循环。每次循环都等待一个来自客户端的连接请求，输出已连接客户端的域名和 IP 地址，并调用 echo 函数为这些客户端服务。在 echo 程序返回后，主程序关闭已连接描述符。一旦客户端和服务器关闭了它们各自的描述符，连接也就终止了。

code/netp/echoserveri.c

```
1   #include "csapp.h"
2
3   void echo(int connfd);
4
5   int main(int argc, char **argv)
6   {
7       int listenfd, connfd, port, clientlen;
8       struct sockaddr_in clientaddr;
9       struct hostent *hp;
10      char *haddrp;
11      if (argc != 2) {
12          fprintf(stderr, "usage: %s <port>\n", argv[0]);
13          exit(0);
14      }
15      port = atoi(argv[1]);
16
17      listenfd = Open_listenfd(port);
18      while (1) {
19          clientlen = sizeof(clientaddr);
20          connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
21
22          /* determine the domain name and IP address of the client */
23          hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
24                             sizeof(clientaddr.sin_addr.s_addr), AF_INET);
25          haddrp = inet_ntoa(clientaddr.sin_addr);
26          printf("server connected to %s (%s)\n", hp->h_name, haddrp);
27
```

```
28          echo(connfd);
29          Close(connfd);
30      }
31      exit(0);
32  }
```

code/netp/echoserveri.c

图 12.20　迭代 echo 服务器的主程序

注意，我们的简单的 echo 服务器一次只能处理一个客户端。这种类型的服务器一次一个地在客户端间迭代，称为**迭代服务器**（iterative server）。在第 13 章中，我们将学习如何建立更加复杂的**并发服务器**（concurrent server），它能够同时处理多个客户端。

最后，图 12.21 展示了 echo 程序的代码，该程序反复读写文本行，直到 rio_readlineb 函数在第 10 行遇到 EOF。

code/netp/echo.c

```
1   #include "csapp.h"
2
3   void echo(int connfd)
4   {
5       size_t n;
6       char buf[MAXLINE];
7       rio_t rio;
8
9       Rio_readinitb(&rio, connfd);
10      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
11          printf("server received %d bytes\n", n);
12          Rio_writen(connfd, buf, n);
13      }
14  }
```

code/netp/echo.c

图 12.21　读和回送文本行的 echo 函数

旁注：在连接中 EOF 意味什么？

EOF 的概念常常使学生们感到迷惑，尤其是在因特网连接的上下文中。首先，我们需要理解其实并没有像 EOF 字符这样的一个东西。进一步来说，EOF 是由内核检测到的一种条件。应用程序在它接收到一个由 read 函数返回的零返回码时，它就会发现出 EOF 条件。对于磁盘文件，当前文件位置超出文件长度时，会发生 EOF。对于因特网连接，当一个进程关闭连接在它的那一端时，会发生 EOF。连接另一端的进程在试图读取流中最后一个字节之后，会检测到 EOF。

12.5　Web 服务器

迄今为止，我们已经讨论了一个简单的 echo 服务器上下文中的网络编程。在这一节里，我们将向你展示如何利用网络编程的基本概念，来创建你自己的虽然小但是功能齐全的 Web 服务器。

12.5.1 Web 基础

Web 客户端和服务器之间的交互用的是一个基于文本的应用级协议，叫做 HTTP（Hypertext Transfer Protocol，超文本传输协议）。HTTP 是一个简单的协议。一个 Web 客户端（就是浏览器）打开一个到服务器的因特网连接，并且请求某些内容。服务器响应所请求的内容，然后关闭连接。浏览器读取这些内容，并把它显示在屏幕上。

Web 服务和常规的文件检索服务（例如 FTP）有什么区别呢？主要的区别是 Web 内容可以用一种叫做 HTML（Hypertext Markup Language，超文本标记语言）的语言来编写。一个 HTML 程序（页）包含指令（标记符），它们告诉浏览器如何显示这页中的各种文本和图形对象。例如，代码

```
<b> Make me bold! </b>
```

告诉浏览器用粗体字类型输出和标记之间的文本。然而，HTML 真正的强大之处在于一个页面可以包含指针（超链接），这些指针可以指向存放在任何因特网主机上的内容。例如，一个格式如下的 HTML 行

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

告诉浏览器高亮显示文本对象“Carnegie Mellon”，并且创建一个超链接，它指向存放在 CMU Web 服务器上叫做 index.html 的 HTML 文件。如果用户单击了这个高亮文本对象，浏览器从 CMU 服务器中请求相应的 HTML 文件，并显示它。

旁注：万维网的起源

万维网是 Tim Berners-Lee 创建的，他是一位在瑞典物理实验室 CERN（欧洲粒子物理研究所）工作的软件工程师。1989 年，Berners-Lee 写了一个内部备忘录，提出了一个分布式超文本系统，它能连接“用链组成的笔记的网（web of notes with links）”。提出这个系统的目的是帮助 CERN 的科学家共享和管理信息。在接下来的两年多里，Berners-Lee 实现了第一个 Web 服务器和 Web 浏览器之后，在 CERN 内部以及其他一些网站中，Web 发展出了小规模的拥护者。1993 年一个关键事件发生了，Marc Andreesen（后来创建了 Netscape）和他在 NCSA 的同事发布了一种图形化的浏览器，叫做 MOSAIC，可以为三种主要的平台所使用：Unix、Windows 和 Macintosh。在 MOSAIC 发布后，对 Web 的兴趣爆发了，Web 网站以每年 10 倍或更高的数量增长。到 2002 年，已经有超过 36 000 000 个世界范围的 Web 网站了（源自 www.netcraft.com 的 Netcraft Web 调查）。

12.5.2 Web 内容

对于 Web 客户端和服务器而言，内容是与一个 MIME（Multipurpose Internet Mail Extensions，多用途的网际邮件扩充协议）类型相关的字节序列。图 12.22 展示了一些常用的 MIME 类型。

MIME类型	描　述
text/html	HTML页面
text/plain	无格式文本
application/postscript	PS文档
image/gif	GIF格式编码的二进制图像
image/jpeg	JPEG 格式编码的二进制图像

图 12.22　MIME 类型示例

Web 服务器以两种不同的方式向客户端提供内容：

- 取一个磁盘文件，并将它的内容返回给客户端。磁盘文件称为**静态内容**（static content），而返回文件给客户端的过程称为**服务静态内容**（serving static content）。
- 运行一个可执行文件，并将它的输出返回给客户端。运行时可执行文件产生的输出称为**动态内容**（dynamic content），而运行程序并返回它的输出到客户端的过程称为**服务动态内容**（serving dynamic content）。

每条由 Web 服务器返回的内容都是和它管理的某个文件相关联的。这些文件中的每一个都有一个惟一的名字，叫做 URL（Universal Resource Locator，通用资源定位符）。例如，URL

```
http://www.aol.com:80/index.html
```

表示因特网主机 www.aol.com 上一个称为/index.html 的 HTML 文件，它是由一个监听 80 端口的 Web 服务器管理的。端口号是可选的，而知名的 HTTP 默认的端口就是 80。可执行文件的 URL 可以在文件名后包括程序参数。“?”字符分隔文件名和参数，而且每个参数都用“&”字符分隔开。例如，URL

```
http://kittyhawk.cmcl.cs.cmu.edu:8000/cgi-bin/adder?15000&213
```

标识了一个叫做/cgi-bin/addr 的可执行文件，会带两个参数字符串 15000 和 213 来调用它。在事务过程中，客户端和服务器使用的是 URL 的不同部分。例如，客户端使用前缀

```
http://www.aol.com:80
```

来决定与哪类服务器联系，服务器在哪儿，以及它监听的端口号是多少。服务器使用后缀

```
/index.html
```

来发现在它文件系统中的文件，并确定请求的是静态内容，还是动态内容。

关于服务器如何解释一个 URL 的后缀，有三点需要理解：

- 确定一个 URL 指向的是静态内容还是动态内容没有标准的规则。每个服务器对它所管理的文件都有自己的规则。一种常见的方法是，确认一组目录，例如 cgi-bin，所有的可执行性文件都必须存放这些目录中。
- 后缀中的最开始的那个“/”不表示 Unix 的根目录。相反，它表示的是被请求内容类型的主目录。例如，可以将一个服务器配置成这样：所有的静态内容存放在目录/usr/httpd/html 下，而所有的动态内容都存放在目录/usr/https/cgi-bin 下。
- 最小的 URL 后缀是“/”字符，所有服务器将其扩展为某个默认的主页，例如/index.html。这解释了为什么简单地在浏览器中键入一个域名就可以取出一个网站的主页。浏览器在 URL 后添加缺失的“/”，并将之传递给服务器，服务器又把“/”扩展到某个默认的文件名。

12.5.3 HTTP 事务

因为 HTTP 是基于在因特网连接上传送的文本行的，我们可以使用 Unix 的 TELNET 程序来和任何因特网上的 Web 服务器执行事务。对于调试在连接上通过文本行来与客户端对话的服务器来说，TELNET 程序是非常便利的。例如，图 12.23 使用 TELNET 向 AOL Web 服务器请求主页。

```
1   unix> telnet www.aol.com 80  Client: open connection to server
2   Trying 205.188.146.23... Telnet prints 3 lines to the terminal
3   Connected to aol.com.
4   Escape character is '^]'.
5   GET / HTTP/1.1             Client: request line
6   host: www.aol.com          Client: required HTTP/1.1 header
7                              Client: empty line terminates headers.
8   HTTP/1.0 200 OK            Server: response line
9   MIME-Version: 1.0          Server: followed by five response headers
10  Date: Mon, 08 Jan 2001 04:59:42 GMT
11  Server: NaviServer/2.0 AOLserver/2.3.3
12  Content-Type: text/html    Server: expect HTML in the response body
13  Content-Length: 42092      Server: expect 42,092 bytes in the response body
14                             Server: empty line terminates response headers
15  <html>                     Server: first HTML line in response body
16   ...                       Server: 766 lines of HTML not shown.
17  </html>                    Server: last HTML line in response body
18  Connection closed by foreign host. Server: closes connection
19  unix>                      Client: closes connection and terminates
```

图 12.23 一个服务静态内容的 HTTP 事务

在第一行，我们从 Unix shell 运行 TELNET，要求它打开一个到 AOL Web 服务器的连接。TELNET 向终端打印三行输出，打开连接，然后等待我们输入文本（第 5 行）。每次我们输入一个文本行，并键入回车键，TELNET 会读取该行，在后面加上回车和换行符号（在 C 的表示中为“\r\n”），并且将这一行发送到服务器。这是和 HTTP 标准相符的，HTTP 标准要求每个文本行都由一个回车和换行符对来结束。为了发起事务，我们输入一个 HTTP 请求（第 5～7 行）。服务器返回 HTTP 响应（第 8～17 行），然后关闭连接（第 18 行）。

HTTP 请求

一个 HTTP 请求的组成是这样的：一个请求行（request line）（第 5 行），后面跟随零个或更多个请求报头（request header）（第 6 行），再跟随一个空的文本行来终止报头列表（第 7 行）。一个请求行的形式是

```
<method><uri><version>
```

HTTP 支持许多不同的方法，包括 GET、POST、OPTIONS、HEAD、PUT、DELETE 和 TRACE。我们将只讨论广为应用的 GET 方法，根据某研究调查，它占了 99%的 HTTP 请求[79]。GET 方法指导服务器生成和返回 URI（Uniform Resource Identifier，统一资源标识符）标识的内容。URI 是相应的 URL 的后缀，包括文件名和可选的参数。[2]

请求行中的<version>字段表明了该请求遵循的 HTTP 版本。最新的 HTTP 版本是 HTTP/1.1[27]。HTTP/1.0 是从 1996 年沿用至今的老版本。HTTP/1.1 定义了一些附加的报头，为诸如缓冲和安全等高级特性提供支持，它还支持一种机制，允许客户端和服务器在同一条持久连接（persistent

---

2 实际上，只有当浏览器请求内容时，才会这样。如果代理服务器请求内容，那么 URI 必须是完整的 URL。

connection）上执行多个事务。在实际中，两个版本是互相兼容的，因为 HTTP/1.0 的客户端和服务器会简单地忽略 HTTP/1.1 的报头。

总地来说，第 5 行的请求行要求服务器取出并返回 HTML 文件/index.html。它也告知服务器请求剩下的部分是 HTTP/1.1 格式的。

请求报头为服务器提供了额外的信息，例如浏览器的商标名，或者浏览器理解的 MIME 类型。请求报头的格式为

```
<header name>: <header data>
```

针对我们的目的，惟一需要关注的报头是 Host 报头（第 6 行）。这个报头在 HTTP/1.1 请求中是需要的，而在 HTTP/1.0 请求中是不需要的。代理缓存（proxy cache）会使用 Host 报头，这个代理缓存有时作为浏览器和管理被请求文件的原始服务器（origin server）的中介。客户端和原始服务器之间，可以有多个代理，即所谓的代理链（proxy chain）。Host 报头中的数据，指示了原始服务器的域名，使得代理链中的代理能够判断它是否可以拥有一个被请求内容的本地缓存的副本。

继续我们图 12.23 中的示例，第 7 行的空文本行（通过在我们的键盘上键入回车键生成的）终止了报头，并指示服务器发送被请求的 HTML 文件。

HTTP 响应

HTTP 响应和 HTTP 请求是相似的。一个 HTTP 响应的组成是这样的：一个响应行（response line）（第 8 行），后面跟随着零个或更多的响应报头（response header）（第 9～13 行），再跟随一个终止报头的空行（第 14 行），再跟随一个响应主体（response body）（第 15～17 行）。一个响应行的格式是

```
<version> <status code> <status message>
```

版本字段描述的是响应所遵循的 HTTP 版本。status code（状态码）是一个三位的正整数，指明对请求的处理。status message（状态消息）给出与错误代码等价的英文描述。图 12.24 列出了一些常见的状态码，以及它们相应的消息。

状态代码	状态消息	描　述
200	成功	处理请求无误
301	永久移动	内容已移动到位置头中指明的主机上
400	错误请求	服务器不能理解请求
403	禁止	服务器无权访问所请求的文件
404	未发现	服务器不能找到所请求的文件
501	未实现	服务器不支持请求的方法
505	HTTP 版本不支持	服务器不支持请求的版本

图 12.24　一些 HTTP 状态码

第 9～13 行的响应报头提供了关于响应的附加信息。针对我们的目的，两个最重要的报头是 Content-Type（第 12 行），它告诉客户端响应主体中内容的 MIME 类型；以及 Content-Length（第 13 行），用来指示响应主体的字节大小。

第 14 行的终止响应报头的空文本行，其后跟随着响应主体，响应主体中包含着被请求的内容。

12.5.4 服务动态内容

如果我们停下来考虑一下，一个服务器是如何向客户端提供动态内容的，就会发现一些问题。例如，客户端如何将程序参数传递给服务器？服务器如何将这些参数传递给它所创建的子进程？服务器如何将子进程生成内容所需要的其他信息传递给子进程？子进程将它的输出发送到哪里？一个称为 CGI（Common Gateway Interface，通用网关接口）的实际标准的出现解决了这些问题。

客户端如何将程序参数传递给服务器？

GET 请求的参数在 URI 中传递。正如我们看到的，一个“？”字符分隔了文件名和参数，而每个参数都用一个“&”字符分隔开。参数中不允许有空格，而必须用字符串“%20”来表示。对其他特殊字符，也存在着相似的编码。

旁注：在 HTTP POST 请求中传递参数

HTTP POST 请求中的参数是在请求主体（request body）中而不是 URI 中传递的。

服务器如何将参数传递给子进程？

在服务器接收一个如下的请求后

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

它调用 fork 来创建一个子进程，并调用 execve 在子进程的上下文中执行/cgi-bin/adder 程序。像 adder 这样的程序，常常被称为 CGI 程序，因为它们遵守 CGI 标准的规则。而且，因为许多 CGI 程序是用 Perl 脚本编写的，所以 CGI 程序也常被称为 CGI 脚本（CGI script）。在调用 execve 之前，子进程将 CGI 环境变量 QUERY_STRING 设置为“15000&213”，adder 程序在运行时可以用 Unix getenv 函数来引用它。

服务器如何将其他信息传递给子进程？

CGI 定义了大量的其他环境变量，一个 CGI 程序在它运行时，可以设置这些环境变量。图 12.25 给出了其中的一部分。

环境变量	描述
QUERY_STRING	程序参数
SERVER_PORT	父进程侦听的端口
REQUEST_METHOD	GET或POST
REMOTE_HOST	客户端的域名
REMOTE_ADDR	客户端的点分十进制IP地址
CONTENT_TYPE	只对POST而言：请求体的MIME类型
CONTENT_LENGTH	只对 POST 而言：请求体的字节大小

图 12.25 CGI 环境变量示例

子进程将它的输出发送到哪里？

一个 CGI 程序将它的动态内容发送到标准输出。在子进程加载并运行 CGI 程序之前，它使用 Unix dup2 函数将标准输出重定向到和客户端相关联的已连接描述符。因此，任何 CGI 程序写到标准输出的东西都会直接到达客户端。

注意，因为父进程不知道子进程生成的内容的类型或大小，所以子进程就要负责生成

Content-type 和 Content-length 响应报头，以及终止报头的空行。

图 12.26 展示了一个简单的 CGI 程序，它对两个参数求和，并返回带结果的 HTML 文件给客户端。图 12.27 展示了一个 HTTP 事务，它从 adder 程序提供动态内容。

code/netp/tiny/cgi-bin/adder.c

```
1   #include "csapp.h"
2
3   int main(void) {
4       char *buf, *p;
5       char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
6       int n1=0, n2=0;
7
8       /* Extract the two arguments */
9       if ((buf = getenv("QUERY_STRING")) != NULL) {
10          p = strchr(buf, '&');
11          *p = '\0';
12          strcpy(arg1, buf);
13          strcpy(arg2, p+1);
14          n1 = atoi(arg1);
15          n2 = atoi(arg2);
16      }
17
18      /* Make the response body */
19      sprintf(content, "Welcome to add.com: ");
20      sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
21      sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
22              content, n1, n2, n1 + n2);
23      sprintf(content, "%sThanks for visiting!\r\n", content);
24
25      /* Generate the HTTP response */
26      printf("Content-length: %d\r\n", strlen(content));
27      printf("Content-type: text/html\r\n\r\n");
28      printf("%s", content);
29      fflush(stdout);
30      exit(0);
31  }
```

code/netp/tiny/cgi-bin/adder.c

图 12.26 对两个整数求和的 CGI 程序

```
1   unix> telnet kittyhawk.cmcl.cs.cmu.edu 8000    Client: open connection
2   Trying 128.2.194.242...
3   Connected to kittyhawk.cmcl.cs.cmu.edu.
4   Escape character is '^]'.
5   GET /cgi-bin/adder?15000&213 HTTP/1.0  Client: request line
6                                          Client: empty line terminates headers
7   HTTP/1.0 200 OK                        Server: response line
```

```
8    Server: Tiny Web                     Server Server: identify server
9    Content-length: 115                  Adder: expect 115 bytes in response body
10    Content-type: text/html             Adder: expect HTML in response body
11                                        Adder: empty line terminates headers
12    Welcome to add.com: THE Internet addition portal. Adder: first HTML line
13    <p>The answer is: 15000 + 213 = 15213   Adder: second HTML line in response body
14    <p>Thanks for visiting!             Adder: third HTML line in response body
15    Connection closed by foreign host. Server: closes connection
16    unix>                               Client: closes connection and terminates
```

图 12.27 一个提供动态 HTML 内容的 HTTP 事务

旁注：在 HTTP POST 请求中传递参数给 CGI 程序

对于 POST 请求，子进程也需要重定向标准输入到已连接描述符。CGI 程序将从标准输入中读取请求主体中的参数。

练习题 12.5

在 11.9 节中，我们警告过你关于在网络应用中使用 C 标准 I/O 函数的危险。然而，图 12.26 中的 CGI 程序却能没有任何问题地使用标准 I/O。为什么呢？

12.6 综合：Tiny Web 服务器

我们通过创建一个虽然小但是功能齐全的称为 Tiny 的 Web 服务器来结束我们对网络编程的讨论。Tiny 是一个有趣的程序。在短短 250 行代码中，它结合了许多我们已经学习到的思想，例如进程控制、Unix I/O、套接字接口和 HTTP。虽然它缺乏一个实际服务器所具备的功能性、稳定性和安全性，但是它足够用来为实际的 Web 浏览器提供静态和动态内容。我们鼓励你研究它，并且自己实现它。将一个实际的浏览器指向你自己的服务器，看着它显示一个复杂的带有文本和图片的 Web 页面，真是非常令人兴奋（甚至对我们这些作者来说，也是如此!）。

Tiny 的 main 程序

图 12.28 展示了 Tiny 的主程序。Tiny 是一个迭代服务器，监听在命令行中确定的端口上的连接请求。在通过调用 open_listenfd 函数打开一个监听套接字以后，Tiny 执行典型的无限服务器循环，反复地接受一个连接请求（第 31 行），执行事务（第 32 行），并关闭连接的它那一端（第 33 行）。

code/netp/tiny/tiny.c

```
1    /*
2     * tiny.c - A simple, iterative HTTP/1.0 Web server that uses the
3     * GET method to serve static and dynamic content.
4     */
5    #include "csapp.h"
6
7    void doit(int fd);
8    void read_requesthdrs(rio_t *rp);
9    int parse_uri(char *uri, char *filename, char *cgiargs);
```

```
10  void serve_static(int fd, char *filename, int filesize);
11  void get_filetype(char *filename, char *filetype);
12  void serve_dynamic(int fd, char *filename, char *cgiargs);
13  void clienterror(int fd, char *cause, char *errnum,
14                   char *shortmsg, char *longmsg);
15
16  int main(int argc, char **argv)
17  {
18      int listenfd, connfd, port, clientlen;
19      struct sockaddr_in clientaddr;
20
21      /* Check command line args */
22      if (argc != 2) {
23          fprintf(stderr, "usage: %s <port>\n", argv[0]);
24          exit(1);
25      }
26      port = atoi(argv[1]);
27
28      listenfd = Open_listenfd(port);
29      while (1) {
30          clientlen = sizeof(clientaddr);
31          connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
32          doit(connfd);
33          Close(connfd);
34      }
35  }
```
code/netp/tiny/tiny.c

图 12.28 Tiny Web 服务器

doit 函数

图 12.29 中的 doit 函数处理一个 HTTP 事务。首先，我们读和解析请求行（第 11~12 行）。注意，我们使用图 11.7 中的 rio_readlineb 函数读取请求行。

code/netp/tiny/tiny.c
```
1   void doit(int fd)
2   {
3       int is_static;
4       struct stat sbuf;
5       char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
6       char filename[MAXLINE], cgiargs[MAXLINE];
7       rio_t rio;
8
9       /* Read request line and headers */
10      Rio_readinitb(&rio, fd);
11      Rio_readlineb(&rio, buf, MAXLINE);
12      sscanf(buf, "%s %s %s", method, uri, version);
13      if (strcasecmp(method, "GET")) {
14          clienterror(fd, method, "501", "Not Implemented",
```

```
15                  "Tiny does not implement this method");
16          return;
17      }
18      read_requesthdrs(&rio);
19
20      /* Parse URI from GET request */
21      is_static = parse_uri(uri, filename, cgiargs);
22      if (stat(filename, &sbuf) < 0) {
23          clienterror(fd, filename, "404", "Not found",
24                      "Tiny couldn't find this file");
25          return;
26      }
27
28      if (is_static) { /* Serve static content */
29          if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
30              clienterror(fd, filename, "403", "Forbidden",
31                          "Tiny couldn't read the file");
32              return;
33          }
34          serve_static(fd, filename, sbuf.st_size);
35      }
36      else { /* Serve dynamic content */
37          if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
38              clienterror(fd, filename, "403", "Forbidden",
39                          "Tiny couldn't run the CGI program");
40              return;
41          }
42          serve_dynamic(fd, filename, cgiargs);
43      }
44  }
```

code/netp/tiny/tiny.c

图 12.29 Tiny doit：处理一个 HTTP 事务

Tiny 只支持 GET 方法。如果客户端请求其他方法（比如 POST），我们发送给它一个错误信息，并返回到主程序（第 13～17 行），主程序随后关闭连接并等待下一个连接请求。否则，我们读并且（像我们将要看到的那样）忽略任何请求报头（第 18 行）。

然后，我们将 URI 解析为一个文件名和一个可能为空的 CGI 参数串，并且我们设置一个标志，表明请求的是静态内容还是动态内容（第 21 行）。如果文件在磁盘上不存在，我们立即发送一个错误信息给客户端，并返回（第 22～26 行）。

最后，如果请求的是静态内容，我们就核实该文件是一个普通文件，而我们是有读权限的（第 29 行）。如果是这样，我们就向客户端提供静态内容。相似地，如果请求的是动态内容，我们就核实该文件是可执行文件（第 37 行），如果是这样，我们就继续，并且提供动态内容（第 42 行）。

clienterror 函数

Tiny 缺乏一个实际服务器的许多错误处理特性。然而，它会检查一些明显的错误，并把它们报告给客户端。图 12.30 中的 clienterror 函数发送一个 HTTP 响应到客户端，在响应行中包含相应的状

态码和状态消息，以及响应主体中的一个 HTML 文件，向浏览器的用户解释这个错误。

code/netp/tiny/tiny.c

```
1   void clienterror(int fd, char *cause, char *errnum,
2                    char *shortmsg, char *longmsg)
3   {
4       char buf[MAXLINE], body[MAXBUF];
5
6       /* Build the HTTP response body */
7       sprintf(body, "<html><title>Tiny Error</title>");
8       sprintf(body, "%s<body bgcolor=""ffffff"">\r\n", body);
9       sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10      sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11      sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);
12
13      /* Print the HTTP response */
14      sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15      Rio_writen(fd, buf, strlen(buf));
16      sprintf(buf, "Content-type: text/html\r\n");
17      Rio_writen(fd, buf, strlen(buf));
18      sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19      Rio_writen(fd, buf, strlen(buf));
20      Rio_writen(fd, body, strlen(body));
21  }
```

code/netp/tiny/tiny.c

图 12.30 Tiny clienterror：向客户端发送一个出错消息

回想一下，HTML 响应应该指明主体中内容的大小和类型。因此，我们选择创建 HTML 内容为一个字符串（第 7～11 行），这样一来我们可以简单地确定它的大小（第 18 行）。还有，请注意我们为所有的输出使用的都是图 11.3 中健壮的 rio_writen 函数。

read_requesthdrs 函数

Tiny 不使用请求报头中的任何信息。它仅仅调用图 12.31 中的 read_requesthdrs 函数来读取并忽略这些报头。注意，终止请求报头的空文本行是由回车和换行符对组成的，我们在第 6 行中检查它。

code/netp/tiny/tiny.c

```
1   void read_requesthdrs(rio_t *rp)
2   {
3       char buf[MAXLINE];
4
5       Rio_readlineb(rp, buf, MAXLINE);
6       while(strcmp(buf, "\r\n"))
7           Rio_readlineb(rp, buf, MAXLINE);
8       return;
9   }
```

code/netp/tiny/tiny.c

图 12.31 Tiny read_requesthdrs：读取并忽略请求报头

parse_uri 函数

Tiny 假设静态内容的主目录就是它的当前目录，而可执行文件的主目录是./cgi-bin。任何包含字符串 cgi-bin 的 URI 都会被认为表示的是对动态内容的请求。默认的文件名是./home.html。

图 12.32 中的 parse_uri 函数实现了这些策略。它将 URI 解析为一个文件名和一个可选的 CGI 参数串。如果请求的是静态内容（第 5 行），我们将清除 CGI 参数串（第 6 行），然后将 URI 转换为一个相对的 Unix 路径名，例如./index.html（第 7～8 行）。如果 URI 是用“/”结尾的（第 9 行），我们将把默认的文件名加在后面（第 10 行）。另一方面，如果请求的是动态内容（第 13 行），我们就会抽取出所有的 CGI 参数（第 14～20 行），并将 URI 剩下的部分转换为一个相对的 Unix 文件名（第 21～22 行）。

code/netp/tiny/tiny.c

```
1   int parse_uri(char *uri, char *filename, char *cgiargs)
2   {
3       char *ptr;
4
5       if (!strstr(uri, "cgi-bin")) {  /* Static content */
6           strcpy(cgiargs, "");
7           strcpy(filename, ".");
8           strcat(filename, uri);
9           if (uri[strlen(uri)-1] == '/')
10              strcat(filename, "home.html");
11          return 1;
12      }
13      else {  /* Dynamic content */
14          ptr = index(uri, '?');
15          if (ptr) {
16              strcpy(cgiargs, ptr+1);
17              *ptr = '\0';
18          }
19          else
20              strcpy(cgiargs, "");
21          strcpy(filename, ".");
22          strcat(filename, uri);
23          return 0;
24      }
25  }
```

code/netp/tiny/tiny.c

图 12.32 Tiny parse_uri:解析一个 HTTP URI

serve_static 函数

Tiny 提供四种不同类型的静态内容：HTML 文件、无格式的文本文件，以及编码为 GIF 和 JPG 格式的图片。这些文件类型占据 Web 上提供的绝大部分静态内容。

图 12.33 中的 serve_static 函数发送一个 HTTP 响应，其主体包含一个本地文件的内容。首先，

我们通过检查文件名的后缀来判断文件类型（第 7 行），并且发送响应行和响应报头给客户端（第 8~12 行）。注意用一个空行终止报头。

code/netp/tiny/tiny.c

```
1   void serve_static(int fd, char *filename, int filesize)
2   {
3       int srcfd;
4       char *srcp, filetype[MAXLINE], buf[MAXBUF];
5
6       /* Send response headers to client */
7       get_filetype(filename, filetype);
8       sprintf(buf, "HTTP/1.0 200 OK\r\n");
9       sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10      sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
11      sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
12      Rio_writen(fd, buf, strlen(buf));
13
14      /* Send response body to client */
15      srcfd = Open(filename, O_RDONLY, 0);
16      srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17      Close(srcfd);
18      Rio_writen(fd, srcp, filesize);
19      Munmap(srcp, filesize);
20  }
21
22  /*
23   * get_filetype - derive file type from file name
24   */
25  void get_filetype(char *filename, char *filetype)
26  {
27      if (strstr(filename, ".html"))
28          strcpy(filetype, "text/html");
29      else if (strstr(filename, ".gif"))
30          strcpy(filetype, "image/gif");
31      else if (strstr(filename, ".jpg"))
32          strcpy(filetype, "image/jpeg");
33      else
34          strcpy(filetype, "text/plain");
35  }
```

code/netp/tiny/tiny.c

图 12.33 Tiny serve_static：为客户端提供静态内容

接着，我们将被请求文件的内容拷贝到已连接描述符 fd，来发送响应主体（第 15~19 行）。这里的代码是比较微妙的，需要仔细研究。第 15 行为读打开了 filename，并获得了它的描述符。在第 16 行，Unix mmap 函数将被请求文件映射到一个虚拟存储器空间。回想我们在第 10.8 节中对 mmap 的讨论，调用 mmap 将文件 srcfd 的前 filesize 个字节映射到一个从地址 srcp 开始的私有只读虚拟存储器区域。

一旦我们将文件映射到存储器，我们就不再需要它的描述符了，所以我们关闭文件（第 17 行）。执行这项任务失败将导致一种潜在的致命的存储器泄漏。第 18 行执行的是到客户端的实际文件传送。rio_writen 函数拷贝从 srcp 位置开始的 filesize 个字节（它们当然已经被映射到了所请求的文件）到客户端的已连接描述符。最后，第 19 行释放了映射的虚拟存储器区域。这对于避免一个潜在的致命的存储器泄漏是很重要的。

serve_dynamic 函数

Tiny 通过派生一个子进程并在子进程的上下文中运行一个 CGI 程序，来提供各种类型的动态内容。

图 12.34 中的 serve_dynamic 函数一开始就向客户端发送一个表明成功的响应行（第 6～7 行），同时还包括带有信息的 Server 报头（第 8～9 行）。CGI 程序负责发送响应的剩余部分。注意，这并不像我们可能希望的那样健壮，因为它没有考虑到 CGI 程序可能会遇到某些错误的可能性。

code/netp/tiny/tiny.c

```
1   void serve_dynamic(int fd, char *filename, char *cgiargs)
2   {
3       char buf[MAXLINE], *emptylist[] = { NULL };
4
5       /* Return first part of HTTP response */
6       sprintf(buf, "HTTP/1.0 200 OK\r\n");
7       Rio_writen(fd, buf, strlen(buf));
8       sprintf(buf, "Server: Tiny Web Server\r\n");
9       Rio_writen(fd, buf, strlen(buf));
10
11      if (Fork() == 0) { /* child */
12          /* Real server would set all CGI vars here */
13          setenv("QUERY_STRING", cgiargs, 1);
14          Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
15          Execve(filename, emptylist, environ); /* Run CGI program */
16      }
17      Wait(NULL); /* Parent waits for and reaps child */
18  }
```

code/netp/tiny/tiny.c

图 12.34 Tiny serve_dynamic：为客户端提供动态内容

在发送了响应的第一部分后，我们会派生一个新的子进程（第 11 行）。子进程用来自请求 URI 的 CGI 参数初始化 QUERY_STRING 环境变量（第 13 行）。注意，一个真正的服务器将还要在此处设置其他的 CGI 环境变量。为了简短，我们省略了这一步。还有，我们注意到 Solaris 系统使用的是 putenv 函数，而不是 setenv 函数。

接下来，子进程重定向它的标准输出到已连接文件描述符（第 14 行），然后加载并运行 CGI 程序（第 15 行）。因为 CGI 程序运行在子进程的上下文中，它能够访问在调用 execve 函数之前就存在的相同的打开文件和环境变量。因此，CGI 程序写到标准输出上的任何东西都将直接送到客户端进程，不会经过任何父进程的干涉。

其间，父进程阻塞在对 wait 的调用中，等待当子进程终止的时候，回收操作系统分配给子进程的资源（第 17 行）。

旁注：处理过早关闭的连接

尽管一个Web服务器的基本功能非常简单，但是我们不想给你一个假象，以为编写一个实际的Web服务器是非常简单的。构造一个运行很长时间而不崩溃的健壮的Web服务器是一件困难的任务，比起在这里我们已经学习了的内容，它要求对Unix系统编程有一个更加深入的理解。例如，如果一个服务器写一个已经被客户端关闭了的连接（比如说，因为你在你的浏览器上单击了“Stop”按钮），那么第一次这样的写会正常返回，但是第二次写就会引起发送SIGPIPE信号，这个信号的默认行为就是终止这个进程。如果捕获或者忽略SIGPIPE信号，那么第二次写操作会返回值-1，并将errno设置为EPIPE。strerr和perror函数将EPIPE错误报告为“Broken pipe”，这是一个迷惑了很多届学生的不太直观的信息。总地来说，一个健壮的服务器必须捕获这些SIGPIPE信号，并且检查write函数调用是否有EPIPE错误。

12.7 小结

每个网络应用都是基于客户端-服务器模型的。根据这个模型，一个应用是由一个服务器和一个或多个客户端组成的。服务器管理资源，以某种方式操作资源，为它的客户端提供服务。客户端-服务器模型中的基本操作是客户端-服务器事务，它是由客户端请求和跟随的服务器响应组成的。

客户端和服务器通过因特网这个全球网络来通信。从一个程序员的观点来看，我们可以把因特网看成是一个全球范围的主机集合，具有以下几个属性：每个因特网都有一个惟一的32位名字，称为它的IP地址；IP地址的集合映射为一个因特网域名的集合；不同因特网主机上的进程能够通过连接互相通信。

客户端和服务器通过使用套接字接口建立连接。套接字是连接的端点，对应用程序来说，连接是以文件描述符的形式出现的。套接字接口提供了打开和关闭套接字描述符的函数。客户端和服务器通过读写这些描述符来实现彼此间的通信。

Web服务器使用HTTP协议和它们的客户端（例如浏览器）彼此通信。浏览器向服务器请求静态或者动态的内容。对静态内容的请求是通过从服务器磁盘取得文件并把它返回给客户端来服务的。对动态内容的请求是通过在服务器上一个子进程的上下文中运行一个程序并将它的输出返回给客户端来服务的。CGI标准提供了一组规则，来管理客户端如何将程序参数传递给服务器，服务器如何将这些参数以及其他信息传递给子进程，以及子进程如何将它的输出发送回客户端。

只用几百行C代码就能实现一个简单但是有功效的Web服务器，它既可以提供静态内容，也可以提供动态内容。

参考文献说明

官方的有关因特网的信息源被保存在一系列的可免费获取的带编号的文档RFC［Requests for Comments，请求注解，Internet标准（草案）］中。在以下网站可获得可搜索的RFC的索引：

```
http://www.rfc-editor.org/rfc.html
```

RFC通常是为因特网基础设施的开发者编写的，因此，对于普通读者来说，往往过于详细了。然而，作为权威信息，没有比它更好的资源了。HTTP/1.1协议记录在RFC 2616中。MIME类型的权威列表保存在：

ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types

关于计算机网络互联有大量好的文献[44，58，84]。伟大的技术作家 W. Richard Stevens 编写了一系列关于诸如高级 Unix 编程[76]、因特网协议[77，78，79]，以及 Unix 网络编程[81，80]之类论题的经典文献。认真学习 Unix 系统编程的学生会想要研究所有这些内容。不幸的是，Stevens 在 1999 年 9 月 1 日逝世。我们会永远纪念他的贡献。

家庭作业

12.6 ◆◆

A. 修改 Tiny 使得它会原样返回每个请求行和请求报头。

B. 使用你喜欢的浏览器向 Tiny 发送一个对静态内容的请求。把 Tiny 的输出记录到一个文件中。

C. 检查 Tiny 的输出，确定你的浏览器使用的 HTTP 的版本。

D. 参考 RFC2616 中的 HTTP/1.1 标准，确定你的浏览器的 HTTP 请求中每个报头的含义。你可以从 www.rfc-editor.org/rfc.html 获得 RFC 2616。

12.7 ◆◆

扩展 Tiny，使得它可以提供 MPG 视频文件。使用一个真正的浏览器来检验你的工作。

12.8 ◆◆

修改 Tiny，使得它在 SIGCHLD 处理程序中回收操作系统分配给 CGI 子进程的资源，而不是显式地等待它们终止。

12.9 ◆◆

修改 Tiny，使得当它服务静态内容时，使用 malloc、rio_readn 和 rio_writen，而不是 mmap 和 rio_writen，来拷贝被请求文件到已连接描述符。

12.10 ◆◆

A. 写出图 12.26 中 CGI adder 函数的 HTML 表单。你的表单应该包括两个文本框，用户将需要相加的两个数字填在这个两个文本框中。你的表单应该使用 GET 方法请求内容。

B. 用这样的方法来检查你的程序：使用一个真正的浏览器向 Tiny 请求表单，向 Tiny 提交填写好的表单，然后显示 adder 生成的动态内容。

12.11 ◆◆

扩展 Tiny，以支持 HTTP HEAD 方法。使用 TELNET 作为 Web 客户端来验证你的工作。

12.12 ◆◆◆

扩展 Tiny，使得它服务以 HTTP POST 方式请求的动态内容。使用你喜欢的 Web 浏览器来验证你的工作。

12.13 ◆◆◆

修改 Tiny，使得它可以干净地处理（而不是终止）在 write 函数试图写一个过早关闭的连接时发生的 SIGPIPE 信号和 EPIPE 错误。

练习题答案

练习题 12.1 答案

十六进制地址	点分十进制地址
0x0	0.0.0.0
0xffffffff	255.255.255.255
0x7f000001	127.0.0.1
0xcdbca079	205.188.160.121
0x400c950d	64.12.149.13
0xcdbc9217	205.188.146.23

练习题 12.2 答案

code/netp/hex2dd.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       struct in_addr inaddr;      /* addr in network byte order */
6       unsigned int addr;          /* addr in host byte order */
7
8       if (argc != 2) {
9           fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
10          exit(0);
11      }
12      sscanf(argv[1], "%x", &addr);
13      inaddr.s_addr = htonl(addr);
14      printf("%s\n", inet_ntoa(inaddr));
15
16      exit(0);
17  }
```

code/netp/hex2dd.c

练习题 12.3 答案

code/netp/dd2hex.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       struct in_addr inaddr;      /* addr in network byte order */
6       unsigned int addr;          /* addr in host byte order */
7
8       if (argc != 2) {
9           fprintf(stderr, "usage: %s <dotted-decimal>\n", argv[0]);
```

```
10            exit(0);
11        }
12
13        if (inet_aton(argv[1], &inaddr) == 0)
14            app_error("inet_aton error");
15        addr = ntohl(inaddr.s_addr);
16        printf("0x%x\n", addr);
17
18        exit(0);
19    }
```
code/netp/dd2hex.c

练习题 12.4 答案

每次我们请求 aol.com 的主机条目时，相应的因特网地址列表以一种不同的、轮转（round-robin）的顺序返回。

```
unix> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13

unix>> ./hostinfo aol.com
official hostname: aol.com
address: 64.12.149.13
address: 205.188.146.23
address: 205.188.160.121

unix>> ./hostinfo aol.com
official hostname: aol.com
address: 205.188.146.23
address: 205.188.160.121
address: 64.12.149.13
```

在不同 DNS 查询中，返回地址的不同顺序称为 DNS 轮转（DNS round-robin），它可以用来对一个大量使用的域名的请求做负载平衡。

练习题 12.5 答案

标准 I/O 能在 CGI 程序里工作的原因是，在子进程中运行的 CGI 程序不需要显式地关闭它的输入输出流。当子进程终止时，内核会自动关闭所有描述符。

CHAPTER 13

并发编程

正如我们在第8章学到的，如果逻辑控制流在时间上重叠，那么它们就是并发（concurrent）的。这种一般现象，称为并发性（concurrency），出现在计算机系统的许多不同层面中。硬件异常处理程序、进程和 Unix 信号处理程序都是大家很熟悉的例子。

到目前为止，我们主要将并发性看做是一种内核用来运行多个应用程序的策略。但是，并发性不仅仅局限于内核，它也可以在应用程序中扮演重要角色。例如，我们已经看到 Unix 信号处理程序如何允许应用响应异步事件，例如用户键入 ctrl-c，或者程序访问虚拟存储器（memory）的一个未定义的区域。应用级并行在其他情况下也是很有用的：

- *在多处理器上进行并行计算*。在只有一个 CPU 的单处理器上，并发流是交替的，在任何时间点上，都只有一个流在 CPU 上实际执行。然而，那些有多个 CPU 的机器，称为多处理器，可以真正地同时执行多个流。被分成并发流的并行应用，在这样的机器上能够运行得快很多。这对大规模数据库和科学应用尤为重要。
- *访问慢速 I/O 设备*。当一个应用正在等待来自慢速 I/O 设备（例如磁盘）的数据到达时，内核会运行其他进程，使 CPU 保持繁忙。每个应用都可以以类似的方式，通过交替执行 I/O 请求和其他有用的工作，来使用并发性。
- *与人交互*。和计算机交互的人要求计算机同时执行多个任务的能力。例如，他们在打印一个文档时，可能想要调整一个窗口的大小。现代视窗系统利用并发性来提供这种能力。每次用户请求某种操作（比如说通过单击鼠标）时，一个独立的并发逻辑流被创建来执行这个操作。
- *通过推迟工作以减少执行时间*。有时，应用程序能够通过推迟其他操作并同时执行它们，利用并发性来降低某些操作的延迟。比如，一个动态存储分配器可以通过推迟与一个运行在较低优先级上的并发"合并"流的合并（coalescing），使用空闲时的 CPU 周期，来降低单个 free 操作的延迟。
- *服务多个网络客户端*。我们在第12章中学习的迭代（iterative）网络服务器是不现实的，因为它们一次只能为一个客户端提供服务。因此，一个慢速的客户端可能会导致服务器拒绝为所有其他客户端服务。对于一个真正的服务器来说，可能期望它每秒为成百上千的客户端提供服务，一个慢速客户端导致拒绝为其他客户端服务，这是不能接受的。一个更好的方法是创建一个并发服务器，它为每个客户端创建各自独立的逻辑流。这就允许服务器同时为多个客户端服务，并且这也避免了慢速客户端独占服务器。

使用应用级并发的应用程序称为并发程序（concurrent program）。现代操作系统提供了三种基本的构造并发程序的方法：

- *进程*。用这种方法，每个逻辑控制流都是一个进程，由内核来调度和维护。因为进程有独立的虚拟地址空间，想要和其他流通信，控制流必须使用某种显式的进程间通信（interprocess communication，IPC）机制。
- *I/O 多路复用*。在这种形式的并发编程中，应用程序在一个进程的上下文中显式地调度它们自己的逻辑流。逻辑流被模型化为状态机，作为数据到达文件描述符的结果，主程序显式地从一个状态转换到另一个状态。因为程序是一个单独的进程，所以所有的流都共享同一个地址空间。
- *线程*。线程是运行在一个单一进程上下文中的逻辑流，由内核进行调度。你可以把线程看

成是其他两种方式的混合体，像进程流一样由内核进行调度，而像 I/O 多路复用流一样共享同一个虚拟地址空间。

本章研究这三种不同的并发编程技术。为了使我们的讨论比较具体，我们始终以同一个应用为例——12.4.9 节中的迭代 echo 服务器的并发版本。

13.1 基于进程的并发编程

构造并发程序最简单的方法就是用进程，使用那些大家都很熟悉的函数，像 fork、exec 和 waitpid。例如，一个构造并发服务器的自然方法就是，在父进程中接受客户端连接请求，然后创建一个新的子进程来为每个新客户端提供服务。

为了了解这是如何工作的，假设我们有两个客户端和一个服务器，服务器正在监听一个监听描述符（比如说是 3）上的连接请求。现在假设服务器接受了客户端 1 的连接请求，并返回一个已连接描述符（比如说是 4），如图 13.1 所示。

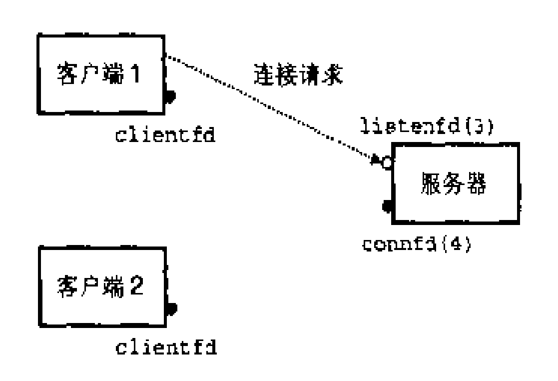


图 13.1 第一步：服务器接受客户端的连接请求

在接受连接请求之后，服务器派生一个子进程，这个子进程获得服务器描述符表的完整拷贝。子进程关闭它的监听描述符 3，而父进程关闭它的已连接描述符 4，因为不再需要这些描述符了。这就得到了图 13.2 中的状态，其中子进程正忙于为客户端提供服务。

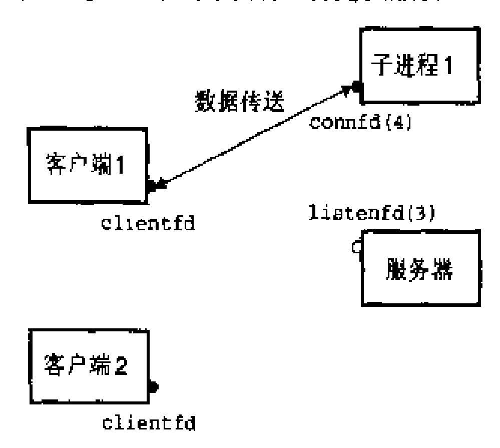


图 13.2 第二步：服务器派生一个子进程为客户端服务

因为父、子进程中的已连接描述符都指向同一个文件表表项，所以父进程关闭它的已连接描述符是至关重要的。否则，将永不会释放已连接描述符 4 的文件表条目，而且由此引起的存储器泄漏

将最终消耗可用的存储器空间，并摧毁系统。

现在，假设在父进程为客户端 1 创建了子进程之后，它接受了一个新的客户端 2 的连接请求，并返回一个新的已连接描述符（比如说是 5），如图 13.3 所示。

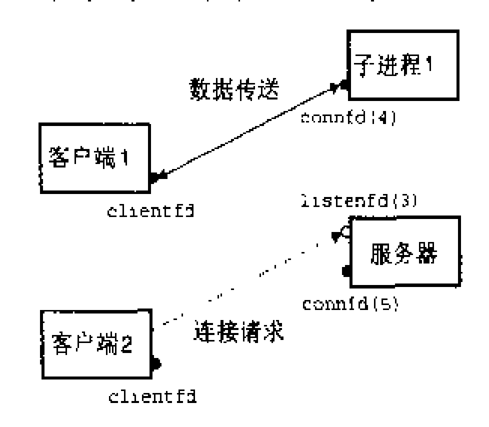


图 13.3 第三步：服务器接受另一个连接请求

然后，父进程又派生一个子进程，使子进程用已连接描述符 5 为它的客户端提供服务，如图 13.4 所示，此时，父进程正在等待下一个连接请求，而两个子进程正在同时为他们各自的客户端提供服务。

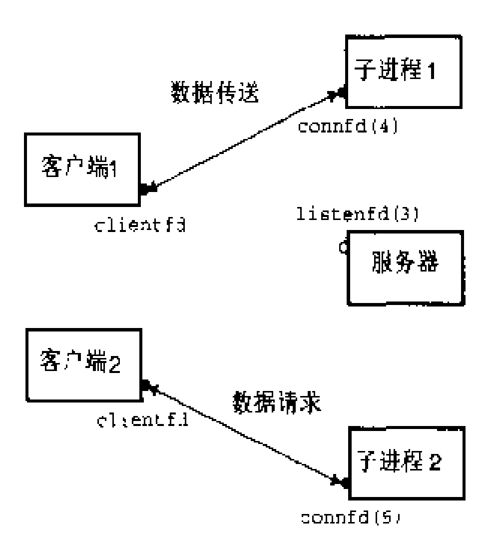


图 13.4 第四步：服务器派生另一个子进程为新的客户端服务

13.1.1 基于进程的并发服务器

图 13.5 展示了一个基于进程的并发 echo 服务器的代码。

code/conc/echoserverp.c

```
1   #include "csapp.h"
2   void echo(int connfd);
3
4   void sigchld_handler(int sig)
5   {
6       while (waitpid(-1, 0, WNOHANG) > 0)
7           ;
8       return;
```

```
9    }
10
11   int main(int argc, char **argv)
12   {
13       int listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
14       struct sockaddr_in clientaddr;
15
16       if (argc != 2) {
17           fprintf(stderr, "usage: %s <port>\n", argv[0]);
18           exit(0);
19       }
20       port = atoi(argv[1]);
21
22       Signal(SIGCHLD, sigchld_handler);
23       listenfd = Open_listenfd(port);
24       while (1) {
25           connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
26           if (Fork() == 0) {
27               Close(listenfd); /* Child closes its listening socket */
28               echo(connfd);    /* Child services client */
29               Close(connfd);   /* Child closes connection with client */
30               exit(0);         /* Child exits */
31           }
32           Close(connfd);       /* Parent closes connected socket (important!) */
33       }
34   }
```

code/conc/echoserverp.c

图 13.5 基于进程的并发 echo 服务器

父进程派生（fork）一个子进程来处理每个新的连接请求。

第 28 行调用的 echo 函数来自于图 12.21。关于这个服务器，有几点重要内容需要说明：

- 首先，通常服务器会运行很长的时间，所以我们需要包括一个 SIGCHLD 处理程序，来回收僵死(zombie)子进程的资源(第 4～9 行)。因为当 SIGCHLD 处理程序执行时，SIGCHLD 信号是阻塞的，而 Unix 信号是不排队的，所以 SIGCHLD 处理程序必须准备好回收多个僵死子进程的资源。
- 其次，父子进程必须关闭它们各自的 connfd（分别为第 32 行和第 29 行）拷贝。就像我们已经提到过的，这对父进程而言尤为重要，它必须关闭它的已连接描述符，以避免存储器泄漏。
- 最后，因为套接字的文件表表项中的引用计数，直到父子进程的 connfd 都关闭了，到客户端的连接才会终止。

13.1.2 关于进程的优劣

对于在父、子进程间共享状态信息，进程有一个非常清晰的模型：共享文件表，但是不共享用户地址空间。有独立的进程地址空间既是优点，也是缺点。这样一来，一个进程不可能不小心覆盖

另一个进程的虚拟存储器，这就消除了许多令人迷惑的错误——这是一个明显的优点。

另一方面，独立的地址空间使得进程共享状态信息变得更加困难。为了共享信息，它们必须使用显式的IPC（进程间通信）机制。基于进程的设计的另一个缺点是，它们往往比较慢，因为进程控制和IPC的开销很高。

旁注：Unix IPC

在本书中，你已经遇到好几个IPC的例子了。第8章中的waitpid函数和Unix信号是基本的IPC机制，它们允许进程发送小消息到同一主机上的其他进程。第12章的套接字是IPC的一种重要形式，它允许不同主机上的进程交换任意的字节流。然而，术语Unix IPC通常指的是所有允许进程和同一台主机上其他进程进行通信的技术，其中包括管道、先进先出（FIFO）、系统V共享存储器，以及系统V信号。这些机制超出了我们的讨论范围。Stevens的著作[80]是很好的参考资料。

练习题 13.1

在图13.5中，并发服务器的第32行上，父进程关闭了已连接描述符后，子进程仍然能够使用该描述符和客户端通信。为什么？

练习题 13.2

如果我们要删除图13.5中关闭已连接描述符的第29行，从没有存储器泄漏的角度来说，代码将仍然是正确的。为什么？

13.2 基于I/O多路复用的并发编程

假设要求你编写一个echo服务器，它也能对用户从标准输入键入的交互命令做出响应。在这种情况下，服务器必须响应两个互相独立的I/O事件：网络客户端发起连接请求；用户在键盘上键入命令行。我们先等待哪个事件呢？没有哪个选择是理想的。如果我们在accept中等待一个连接请求，我们就不能响应输入的命令。类似地，如果我们在read中等待一个输入命令，我们就不能响应任何连接请求。

针对这种困境的一个解决办法就是*I/O多路复用*（I/O multiplexing）技术。基本的思路就是使用select函数，要求内核挂起进程，只有在一个或多个I/O事件发生后，才将控制返回给应用程序，就像在下面的示例中一样：

- 当集合{0，4}中任意描述符准备好读时返回。
- 当集合{1，2，7}中任意描述符准备好写时返回。
- 如果在等待一个I/O事件发生时过了152.13秒，就超时。

select是一个复杂的函数，有许多不同的使用模式。我们将只讨论第一种模式：等待一组描述符准备好读。全面的讨论请参考[76，81]。

```
#include <unistd.h>
#include <sys/types.h>
```

```
int select(int n, fd_set *fdset, NULL, NULL, NULL);
```
返回已准备好的描述符的非零个数，若出错则为-1。

```
FD_ZERO(fd_set *fdset);            /* Clear all bits in fdset */
FD_CLR(int fd, fd_set *fdset);     /* Clear bit fd in fdset */
FD_SET(int fd, fd_set *fdset);     /* Turn on bit fd in fdset */
FD_ISSET(int fd, fd_set *fdset);   /* Is bit fd in fdset turned on? */
```
处理描述符集合的宏。

select 函数处理类型为 fd_set 的集合，也叫做描述符集合。逻辑上，我们将描述符集合看成一个大小为 n 的位掩码：

$$b_{n-1},\cdots,b_1,b_0。$$

每个位 b_k 对应于描述符 k。当且仅当有 $b_k=1$，描述符 k 才表明是描述符集合的一个元素。只允许你对描述符集合做三件事：分配它们；将一个此种类型的变量赋值给另一个变量；用 **FD_ZERO**、**FD_SET**、**FD_CLR** 和 **FD_ISSET** 宏指令来修改和检查它们。

针对我们的目的，select 函数有两个输入：一个称为读集合的描述符集合（fdset）和该读集合的元素量（n）。select 函数会一直阻塞，直到读集合中至少有一个描述符准备好可以读。当且仅当一个从该描述符读取一个字节的请求不会阻塞时，描述符 k 就表示准备好可以读了。作为一个副作用，select 修改了参数 fdset 指向的 fd_set，指明读集合中一个称为准备好集合（ready set）的子集，这个集合是由读集合中准备好可以读了的描述符组成的。函数返回的值指明了准备好集合的元素量。注意，由于这个副作用，我们必须在每次调用 select 时都更新读集合。

理解 select 的最好办法是研究一个具体例子。图 13.6 展示了我们可以如何利用 select 来实现一个迭代 echo 服务器，它也可以接受标准输入上的用户命令。

code/conc/select.c

```
1   #include "csapp.h"
2   void echo(int connfd);
3   void command(void);
4
5   int main(int argc, char **argv)
6   {
7       int listenfd, connfd, port, clientlen = sizeof(struct sockaddr_in);
8       struct sockaddr_in clientaddr;
9       fd_set read_set, ready_set;
10
11      if (argc != 2) {
12          fprintf(stderr, "usage: %s <port>\n", argv[0]);
13          exit(0);
14      }
15      port = atoi(argv[1]);
16      listenfd = Open_listenfd(port);
17
18      FD_ZERO(&read_set);
```

```
19        FD_SET(STDIN_FILENO, &read_set);
20        FD_SET(listenfd, &read_set);
21
22        while (1) {
23            ready_set = read_set;
24            Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25            if (FD_ISSET(STDIN_FILENO, &ready_set))
26                command(); /* read command line from stdin */
27            if (FD_ISSET(listenfd, &ready_set)) {
28                connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
29                echo(connfd); /* echo client input until EOF */
30            }
31        }
32    }
33
34    void command(void) {
35        char buf[MAXLINE];
36        if (!Fgets(buf, MAXLINE, stdin))
37            exit(0); /* EOF */
38        printf("%s", buf); /* Process the input command */
39    }
```

code/conc/select.c

图 13.6 使用 I/O 多路复用的 echo 服务器

服务器使用 select 等待监听描述符上的连接请求和标准输入上的命令。

一开始，我们用图 12.7 中的 open_listenfd 函数打开一个监听描述符（第 16 行），然后使用 FD_ZERO 创建一个空的读集合：

	listenfd			stdin
	3	2	1	0
read_set (∅):	0	0	0	1

接下来，在第 19～20 行中，我们定义由描述符 0（标准输入）和描述符 3（监听描述符）组成的读集合：

	listenfd			stdin
	3	2	1	0
read_set({0,3}):	0	0	0	1

在这里，我们开始典型的服务器循环。但是我们不调用 accept 函数来等待一个连接请求，而是调用 select 函数，这个函数会一直阻塞，直到监听描述符或者标准输入准备好可以读（第 24 行）。例如，下面是当用户敲击回车键，因此使得标准输入描述符变为可读时，select 会返回的 ready_set 的值：

	listenfd			stdin
	3	2	1	0
read_set({0}):	0	0	0	1

一旦 select 返回，我们就用 FD_ISSET 宏指令来判断哪个描述符准备好可以读了。如果是标准输入准备好了（第 25 行），我们就调用 command 函数，该函数在返回到主程序前，会读、解析和响应命令。如果是监听描述符准备好了（第 27 行），我们就调用 accept 来得到一个已连接描述符，然后调用图 12.21 中的 echo 函数，它会将来自客户端的每一行又回送回去，直到客户端关闭它的连接。

虽然这个程序是使用 select 的一个很好示例，但是它仍然留下了一些问题待解决。问题是一旦它连接到某个客户端，就会连续回送输入行，直到客户端关闭它的连接端。因此，如果你键入一个命令到标准输入，你将不会得到响应，直到服务器和客户端之间结束。一个更好的方法是更细粒度的多路复用，服务器每次循环（至多）回送一个文本行（参见练习题 13.3）。

13.2.1 基于 I/O 多路复用的并发事件驱动服务器

I/O 多路技术可以用做并发**事件驱动**（event-driven）程序的基础，在事件驱动中，流是作为某种事件的结果前进的。一般概念是将逻辑流模型化为状态机。不严格地说，一个**状态机**（state machine）就是一组**状态**（state）、**输入事件**（input event）和**转移**（transition），其中转移就是将状态和输入事件映射到状态。每个转移都将一对（输入状态和输入事件）映射到一个输出状态。**自循环**（self-loop）是同一输入和输出状态之间的转移。通常把状态机画成有向图，其中节点表示状态，有向弧表示转移，而弧上的标号表示输入事件。一个状态机从某种初始状态开始执行。每个输入事件都会引发一个从当前状态到下一状态的转移。

对于每个新的客户端 k，基于 I/O 多路复用的并发服务器会创建一个新的状态机 s_k，并将它和已连接描述符 d_k 联系起来。如图 13.7 所示，每个状态机 s_k 都有一个状态（"等待描述符 d_k 准备好可读"）、一个输入事件（"描述符 d_k 准备好可以读了"）和一个转移（"从描述符 d_k 读一个文本行"）。

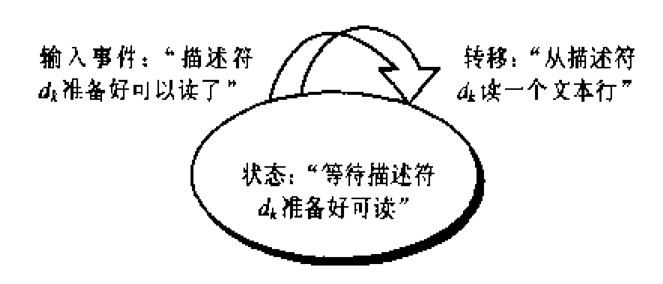


图 13.7 并发事件驱动 echo 服务器中逻辑流的状态机

服务器使用 I/O 多路复用，借助 select 函数，检测输入事件的发生。当每个已连接描述符准备好可读时，服务器就为相应的状态机执行转移，在这里就是从描述符读和写回一个文本行。

图 13.8 展示了一个基于 I/O 多路复用的并发事件驱动服务器的完整示例代码。活动客户端的集合维护在一个 pool（池）结构里（第 3～11 行）。在通过调用 init_pool 初始化池（第 28 行）之后，服务器进入一个无限循环。在每次循环中，服务器调用 select 函数来检测两种不同类型的输入事件：来自一个新客户端的连接请求到达；一个已存在的客户端的已连接描述符准备好可以读了。当一个连接请求到达时（第 35 行），服务器打开连接（第 36 行），并调用 add_client 函数，将该客户端添

加到池里（第 37 行）。最后，服务器调用 check_client 函数，把来自每个准备好的已连接描述符的一个文本行回送回去（第 41 行）。

code/conc/echoservers.c

```
1   #include "csapp.h"
2
3   typedef struct {          /* represents a pool of connected descriptors */
4       int maxfd;            /* largest descriptor in read_set */
5       fd_set read_set;      /* set of all active descriptors */
6       fd_set ready_set;     /* subset of descriptors ready for reading */
7       int nready;           /* number of ready descriptors from select */
8       int maxi;             /* highwater index into client array */
9       int clientfd[FD_SETSIZE];        /* set of active descriptors */
10      rio_t clientrio[FD_SETSIZE];     /* set of active read buffers */
11  } pool;
12
13  int byte_cnt = 0; /* counts total bytes received by server */
14
15  int main(int argc, char **argv)
16  {
17      int listenfd, connfd, port, clientlen = sizeof(struct sockaddr_in);
18      struct sockaddr_in clientaddr;
19      static pool pool;
20
21      if (argc != 2) {
22          fprintf(stderr, "usage: %s <port>\n", argv[0]);
23          exit(0);
24      }
25      port = atoi(argv[1]);
26
27      listenfd = Open_listenfd(port);
28      init_pool(listenfd, &pool);
29      while (1) {
30          /* Wait for listening/connected descriptor(s) to become ready */
31          pool.ready_set = pool.read_set;
32          pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34          /* If listening descriptor ready, add new client to pool */
35          if (FD_ISSET(listenfd, &pool.ready_set)) {
36              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
37              add_client(connfd, &pool);
38          }
39
40          /* Echo a text line from each ready connected descriptor */
41          check_clients(&pool);
```

```
42          }
43    }
```

code/conc/echoservers.c

图 13.8 基于 I/O 多路复用的并发 echo 服务器

每次服务器循环都回送来自每个准备好的描述符的文本行。

init_pool 函数（图 13.9）初始化客户端池。clientfd 数组表示已连接描述符的集合，其中-1 表示一个可用的槽位。初始时，已连接描述符集合是空的（第 5~7 行），而且监听描述符是 select 读集合中惟一的描述符（第 10~12 行）。

code/conc/echoservers.c

```
1   void init_pool(int listenfd, pool *p)
2   {
3       /* Initially, there are no connected descriptors */
4       int i;
5       p->maxi = -1;
6       for (i=0; i< FD_SETSIZE; i++)
7           p->clientfd[i] = -1;
8
9       /* Initially, listenfd is only member of select read set */
10      p->maxfd = listenfd;
11      FD_ZERO(&p->read_set);
12      FD_SET(listenfd, &p->read_set);
13  }
```

code/conc/echoservers.c

图 13.9 init_pool：初始化活动客户端池

add_client（图 13.10）函数添加一个新的客户端到活动客户端池。在 clientfd 数组中找到一个空位后，服务器将这个已连接描述符添加到数组中，并初始化相应的 Rio 读缓冲区，使得我们能够对这个描述符调用 rio_readlineb（第 8~9 行）。然后，我们将这个已连接描述符添加到 select 读集合（第 12 行），并更新该池的一些全局属性。maxfd 变量（第 15~16 行）记录了 select 的最大文件描述符。maxi 变量（第 17~18 行）记录的是 clientfd 数组的最大索引，这样 check_clients 函数就无需搜索整个数组了。

code/conc/echoservers.c

```
1   void add_client(int connfd, pool *p)
2   {
3       int i;
4       p->nready--;
5       for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
6           if (p->clientfd[i] < 0) {
7               /* Add connected descriptor to the pool */
8               p->clientfd[i] = connfd;
9               Rio_readinitb(&p->clientrio[i], connfd);
```

```
10
11              /* Add the descriptor to descriptor set */
12              FD_SET(connfd, &p->read_set);
13
14              /* Update max descriptor and pool highwater mark */
15              if (connfd > p->maxfd)
16                  p->maxfd = connfd;
17              if (i > p->maxi)
18                  p->maxi = i;
19              break;
20          }
21      if (i == FD_SETSIZE) /* Couldn't find an empty slot */
22          app_error("add_client error: Too many clients");
23  }
```

code/conc/echoservers.c

图 13.10 add_client：添加一个新的客户端连接到池中

check_clients（图 13.11）函数回送来自每个准备好的已连接描述符的一个文本行。如果我们成功地从描述符读取了一个文本行，那么我们就将该文本行回送到客户端（第 15～18 行）。注意，在第 15 行我们维护着一个从所有客户端接收到的全部字节的累计值。如果因为客户端关闭它的连接端，我们检测到 EOF，那么我们将关闭我们这边的连接端（第 23 行），并从池中清除掉这个描述符（第 24～25 行）。

code/conc/echoservers.c

```
1   void check_clients(pool *p)
2   {
3       int i, connfd, n;
4       char buf[MAXLINE];
5       rio_t rio;
6
7       for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8           connfd = p->clientfd[i];
9           rio = p->clientrio[i];
10
11          /* If the descriptor is ready, echo a text line from it */
12          if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
13              p->nready--;
14              if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
15                  byte_cnt += n;
16                  printf("Server received %d (%d total) bytes on fd %d\n",
17                         n, byte_cnt, connfd);
18                  Rio_writen(connfd, buf, n);
19              }
20
```

```
21              /* EOF detected, remove descriptor from pool */
22              else {
23                  Close(connfd);
24                  FD_CLR(connfd, &p->read_set);
25                  p->clientfd[i] = -1;
26              }
27          }
28      }
29  }
```

code/conc/echoservers.c

图 13.11 check_clients：为准备好的客户端连接服务

根据图 13.7 中的有限状态模型，select 函数检测到输入事件，而 add_client 函数创建一个新的逻辑流（状态机）。check_clients 函数通过回送输入行来执行状态转移，而且当客户端完成文本行发送时，它还要删除这个状态机。

13.2.2 I/O 多路复用技术的优劣

图 13.8 中的服务器提供了一个很好的基于 I/O 多路复用的事件驱动编程的优缺点示例。事件驱动设计的一个优点是，它比基于进程的设计给了程序员更多的对程序行为的控制。例如，我们可以设想编写一个事件驱动的并发服务器，为某些客户端提供它们需要的服务，而这对于基于进程的并发服务器来说，是很困难的。

另一个优点是，一个基于 I/O 多路复用的事件驱动服务器是运行在单一进程上下文中的，因此每个逻辑流都能访问该进程的全部地址空间。这使得在流之间共享数据变得很容易。作为单一进程运行的一个相关优点是，你可以利用熟悉的调试工具（例如 GDB）来调试你的并发服务器，就像对顺序程序那样。最后，事件驱动设计常常比基于进程的设计要明显地高效得多，因为它们不要求有进程上下文切换来调度新的流。

事件驱动设计一个明显的缺点就是编码复杂。例如，我们的事件驱动的并发 echo 服务器需要的代码比基于进程的服务器多三倍，并且很不幸，随着并发性粒度的减小，复杂性还会上升。这里的粒度是指每个逻辑流每次时间片执行的指令数目。例如，在我们的示例并发服务器中，并发粒度就是读一个完整的文本行所需要的指令数目。只要某个逻辑流正忙于读一个文本行，其他逻辑流就不可能有进展。对我们的例子而言这就很好了，但是它使得我们的事件驱动服务器在“故意只发送部分文本行然后就停止”的恶意客户端的攻击面前显得很脆弱。修改事件驱动服务器来处理部分文本行不是一个简单的任务，但是基于进程的设计却能处理得很好，而且是自动处理的。

练习题 13.3

在大多数的 Unix 系统里，在标准输入上键入 ctrl-d 表示 EOF。如果你在图 13.6 中的程序阻塞在对 select 的调用上时，键入 ctrl-d 会发生什么？

练习题 13.4

图 13.8 所示的服务器中，我们在每次调用 select 之前都立即小心地重新初始化 pool.ready_set 变量。为什么？

13.3 基于线程的并发编程

到目前为止，我们已经看到了两种创建并发逻辑流的方法。在第一种方法中，我们为每个流使用了单独的进程。内核会自动调度每个进程。每个进程有它自己的私有地址空间，这使得流共享数据很困难。在第二种方法中，我们创建自己的逻辑流，并利用I/O多路复用来显式地调度流。因为只有一个进程，所有的流共享整个地址空间。这一节介绍第三种方法——基于线程——它是这两种方法的混合。

一个线程（thread）就是运行在一个进程上下文中的一个逻辑流。迄今在本书里，我们的程序都是由一个进程中一个线程组成的。但是现代系统也允许我们编写一个进程里同时运行多个线程的程序。线程由内核自动调度。每个线程都有它自己的线程上下文（thread context），包括一个惟一的整数线程ID（Thread ID，TID）、栈、栈指针、程序计数器、通用目的寄存器和条件码。所有的运行在一个进程里的线程共享该进程的整个虚拟地址空间。

基于线程的逻辑流结合了基于进程和基于I/O多路复用的流的特性。同进程一样，线程由内核自动调度，并且内核通过一个整数ID来识别线程。同基于I/O多路复用的流一样，多个线程运行在单一进程的上下文中，因此共享这个进程虚拟地址空间的整个内容，包括它的代码、数据、堆、共享库和打开的文件。

13.3.1 线程执行模型

多个线程的执行模型在某些方面和多进程的执行模型是很相似的。思考一下图13.12中的示例。每个进程开始生命周期时都是单一线程，这个线程称为**主线程**（main thread）。在某一时刻，主线程创建一个**对等线程**（peer thread），从这个时间点开始，两个线程就并发运行。最后，因为主线程执行一个慢速系统调用，例如read或者sleep，或者因为它被系统的间隔计时器中断，控制就会通过上下文切换传递到对等线程。在控制传递回主线程前，对等线程会执行一段时间，依次类推。

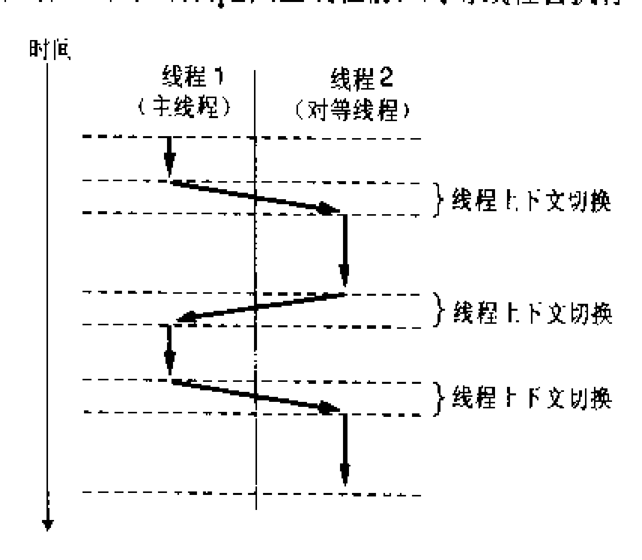


图13.12 并发线程的执行

在一些重要的方面，线程执行是不同于进程的。因为一个线程的上下文要比一个进程的上下文小得多，线程的上下文切换要比进程的上下文切换快得多。另一个不同就是线程，不像进程那样，不是按照严格的父子层次来组织的。和一个进程相关的线程组成一个对等（线程）池（a pool of peers），

独立于其他线程创建的线程。主线程和其他线程的区别仅在于它总是进程中第一个运行的线程。对等（线程）池概念的主要影响是，一个线程可以杀死它的任何对等线程，或者等待它的任意对等线程终止。进一步来说，每个对等线程都能读写相同的共享数据。

13.3.2 Posix 线程

Posix 线程（Pthreads）是在 C 程序中处理线程的一个标准接口。它最早出现在 1995 年，而且在大多数 Unix 系统上都可用。Pthreads 定义了大约 60 个函数，允许程序创建、杀死和回收线程，与对等线程安全地共享数据，还可以通知对等线程系统状态的变化。

图 13.13 展示了一个简单的 Pthreads 程序。主线程创建一个对等线程，然后等待它的终止。对等线程输出“Hello, world!\n”并且终止。当主线程检测到对等线程终止后，它就通过调用 exit 终止该进程。

code/conc/hello.c

```
1   #include "csapp.h"
2   void *thread(void *vargp);
3
4   int main()
5   {
6       pthread_t tid;
7       Pthread_create(&tid, NULL, thread, NULL);
8       Pthread_join(tid, NULL);
9       exit(0);
10  }
11
12  void *thread(void *vargp) /* thread routine */
13  {
14      printf("Hello, world!\n");
15      return NULL;
16  }
```

code/conc/hello.c

图 13.13 hello.c: Pthreads “Hello, world!” 程序

这是我们看到的第一个线程化的程序，所以让我们仔细地解析它。线程的代码和本地数据被封装在一个线程例程（thread routine）中。正如第二行里的原型所示，每个线程例程都以一个通用指针作为输入，并返回一个通用指针。如果你想传递多个参数给线程例程，那么你应该将参数放到一个结构中，并传递一个指向该结构的指针。相似地，如果你想要线程例程返回多个参数，你可以返回一个指向一个结构的指针。

第 4 行标出了主线程代码的开始。主线程声明了一个本地变量 tid，它可以用来存放对等线程的线程 ID（第 6 行）。主线程通过调用 pthread_create 函数创建一个新的对等线程（第 7 行）。当对 pthread_create 的调用返回时，主线程和新创建的对等线程并发运行，并且 tid 包含新线程的 ID。通过调用 pthread_join，主线程等待对等线程终止（第 8 行）。最后，主线程调用 exit（第 9 行），终止当时运行在这个进程中的所有线程（在这个示例中就只有主线程）。

第 12~16 行定义了对等线程的线程例程。它只打印一个字符串，然后就通过在第 15 行执行

return 语句来终止对等线程。

13.3.3 创建线程

线程通过调用 pthread_create 函数来创建其他线程。

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
     func *f, void *arg);
```

若成功则返回 0，若出错则为非零。

pthread_create 函数创建一个新的线程，并带着一个输入变量 arg，在新线程的上下文中运行线程例程 f。能用 attr 参数来改变新创建线程的默认属性。改变这些属性已超出我们学习的范围，并且在我们的示例中，我们总是用一个空的 attr 参数来调用 pthread_create 函数。

当 pthread_create 返回时，参数 tid 包含新创建线程的 ID。新线程可以通过调用 pthread_self 函数来获得它自己的线程 ID。

```
#include <pthread.h>

pthread_t pthread_self(void);
```

返回调用者的线程 ID。

13.3.4 终止线程

一个线程是以下列方式之一来终止的：

- 当顶层的线程例程返回时，线程会隐式地终止。
- 通过调用 pthread_exit 函数，线程会显式地终止，该函数会返回一个指向返回值 thread_return 的指针。如果主线程调用 pthread_exit，它会等待所有其他对等线程终止，然后再终止主线程和整个进程，返回值为 thread_return。

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```

若成功则返回 0，若出错则为非零。

- 某个对等线程调用 Unix 的 exit 函数，该函数终止进程以及所有与该进程相关的线程。
- 另一个对等线程通过带调用当前线程 ID 来的 pthread_cancle 函数来终止当前线程。

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

若成功则返回 0，若出错则为非零。

13.3.5 回收已终止线程的资源

线程通过调用 pthread_join 函数等待其他线程终止。

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

若成功则返回 0，若出错则为非零。

pthread_join 函数会阻塞，直到线程 tid 终止，将线程例程返回的（void*）指针赋值为 thread_return 指向的位置，然后回收已终止线程占用的所有存储器资源。

注意，和 Unix 的 wait 函数不同的，pthread_join 函数只能等待一个指定的线程终止。没有办法让 pthread_wait 等待任意一个线程终止。这使得我们的代码更加复杂，因为它迫使我们去使用其他一些不那么直观的机制来检测进程的终止。实际上，Stevens 在[81]中就很有说服力地论证了这是一个错误。

13.3.6 分离线程

在任何一个时间点上，线程是可结合的（joinable）或者是分离的（detached）。一个可结合的线程能够被其他线程收回其资源和杀死。在被其他线程回收之前，它的存储器资源（例如栈）是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的。它的存储器资源在它终止时由系统自动释放。

默认情况下，线程被创建成可结合的。为了避免存储器泄漏，每个可结合线程都应该要么被其他线程显式地收回，要么通过调用 pthread_detach 函数被分离。

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

若成功则返回 0，若出错则为非零。

pthread_detach 函数分离可结合线程 tid。线程能够通过以 pthread_self()为参数的 pthread_detach 调用来分离它们自己。

尽管我们的一些例子会使用可结合线程，但是在现实程序中，有理由要使用分离的线程。例如，一个高性能 Web 服务器可能在每次收到 Web 浏览器的连接请求时都创建一个新的对等线程。因为每个连接都是由一个单独的线程独立处理的，所以对于服务器而言，就很没有必要——实际上也不愿意——显式地等待每个对等线程终止。在这种情况下，每个对等线程都应该在它开始处理请求之前，分离它自身，这样就能在它终止后，回收它的存储器资源了。

13.3.7 初始化线程

pthread_once 函数允许你初始化与线程例程相关的状态。

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
```

总是返回 0。

once_control 变量是一个全局或者静态变量，总是被初始化为 PTHREAD_ONCE_INIT。当你第一次用参数 once_control 调用 pthread_once 时，它调用 init_routine，这是一个没有输入参数，也不返回什么的函数。接下来的以 once_control[1] 为参数的 pthread_once 调用不做任何事情。无论何时，当你需要动态初始化多个线程共享的全局变量时，pthread_once 函数是很有用的。我们将在 13.6 节里看到一个示例。

13.3.8 一个基于线程的并发服务器

图 13.14 展示了基于线程的并发 echo 服务器的代码。整体结构类似于基于进程的设计。主线程不断地等待连接请求，然后创建一个对等线程处理该请求。虽然代码看似简单，但是有几个普遍而且有些微妙的问题需要我们更仔细地看一看。第一个问题是当我们调用 pthread_create 时，如何将已连接描述符传递给对等线程。最明显的方法就是传递一个指向这个描述符的指针，就像下面这样

```
connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);
```

然后，我们让对等线程间接引用这个指针，并将它赋值给一个局部变量，如下所示

```
void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    ...
}
```

然而，这样可能会出错，因为它在对等线程的赋值语句和主线程的 accept 语句间引入了竞争(race)。如果赋值语句在下一个 accept 之前完成，那么对等线程中的局部变量 connfd 就得到正确的描述符值。然而，如果赋值语句是在 accept 之后才完成的，那么对等线程中的局部变量 connfd 就得到下一次连接的描述符值。那么不幸的结果就是，现在两个线程在同一个描述符上执行输入和输出。为了避免这种潜在的致命竞争，我们必须将每个 accept 返回的已连接描述符分配到它自己的动态分配的存储器块，如第 20～21 行所示。我们会在 13.7.4 节中回过来讨论竞争的问题。

code/conc/echoservert.c

```
1   #include "csapp.h"
2
3   void echo(int connfd);
4   void *thread(void *vargp);
5
6   int main(int argc, char **argv)
7   {
8       int listenfd, *connfdp, port, clientlen=sizeof(struct sockaddr_in);
9       struct sockaddr_in clientaddr;
10      pthread_t tid;
11
12      if (argc != 2) {
13          fprintf(stderr, "usage: %s <port>\n", argv[0]);
```

1 原文为 pthread_once。——译者

```
14          exit(0);
15      }
16      port = atoi(argv[1]);
17
18      listenfd = Open_listenfd(port);
19      while (1) {
20          connfdp = Malloc(sizeof(int));
21          *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
22          Pthread_create(&tid, NULL, thread, connfdp);
23      }
24  }
25
26  /* thread routine */
27  void *thread(void *vargp)
28  {
29      int connfd = *((int *)vargp);
30      Pthread_detach(pthread_self());
31      Free(vargp);
32      echo(connfd);
33      Close(connfd);
34      return NULL;
35  }
```

code/conc/echoservert.c

图 13.14 基于线程的并发 echo 服务器

另一个问题是在线程例程中避免存储器泄漏。既然我们不显式地收回线程，我们就必须分离每个线程，使得它的存储器资源在它终止时能够被收回（第 30 行）。更进一步，我们必须小心释放主线程分配的存储器块（第 31 行）。

练习题 13.5

在图 13.5 中基于进程的服务器中，我们在两个位置小心地关闭了已连接描述符：父进程和子进程。然而，在图 13.14 中基于线程的服务器中，我们只在一个位置关闭了已连接描述符：对等线程。为什么？

13.4 多线程程序中的共享变量

从一个程序员的角度来看，线程很有吸引力的一个方面就是多个线程很容易共享相同的程序变量。然而，这种共享也是很棘手的。为了编写正确的多线程程序，我们必须对所谓的共享以及它是如何工作的有很清楚的了解。

为了理解 C 程序中的一个变量是否是共享的，有一些基本的问题要解答：线程的基础存储器模型是什么？根据这个模型，变量实例是如何映射到存储器的？最后，有多少线程引用这些实例？一

个变量是共享的，当且仅当多个线程引用这个变量的某个实例。

为了让我们对共享的讨论具体化，我们将使用图13.15中的程序作为一个运行示例。尽管有些人为的痕迹，但是它仍然值得研究，因为它说明了关于共享的许多细微之处。示例程序由一个创建了两个对等线程的主线程组成。主线程传递一个惟一的ID给每个对等线程，每个对等线程利用这个ID输出一条个性化的信息，以及调用该线程例程的全部次数的数值。

code/conc/sharing.c

```
1   #include "csapp.h"
2   #define N 2
3   void *thread(void *vargp);
4
5   char **ptr; /* global variable */
6
7   int main()
8   {
9       int i;
10      pthread_t tid;
11      char *msgs[N] = {
12          "Hello from foo",
13          "Hello from bar"
14      };
15
16      ptr = msgs;
17      for (i = 0; i < N; i++)
18          Pthread_create(&tid, NULL, thread, (void *)i);
19      Pthread_exit(NULL);
20  }
21
22  void *thread(void *vargp)
23  {
24      int myid = (int)vargp;
25      static int cnt = 0;
26      printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
27  }
```

code/conc/sharing.c

图13.15 说明共享不同方面的示例程序

13.4.1 线程存储器模型

一组并发线程运行在一个进程的上下文中。每个线程都有它自己独立的线程上下文，包括线程ID、栈、栈指针、程序计数器、条件代码和通用目的寄存器值。每个线程和其他线程一起共享进程上下文的剩余部分。这包括整个用户虚拟地址空间，它是由只读文本（代码）、读/写数据、堆以及所有的共享库代码和数据区域组成的。线程也共享同样的打开文件的集合。

从实际操作的角度来说，让一个线程去读或写另一个线程的寄存器值是不可能的。另一方面，

任何线程都可以访问共享虚拟存储器的任意位置。如果某个线程修改了一个存储器位置，那么其他每个线程最终都能在它读这个位置时发现这个变化。因此，寄存器是从不共享的，而虚拟存储器总是共享的。

各自独立的线程栈的存储器模型不是那么整齐清楚的。这些栈被保存在虚拟地址空间的栈区域中，并且通常是被它们相应的线程独立地访问的。我们说通常而不是总是，是因为不同的线程栈是不对其他线程设防的。所以，如果一个线程不知何故得到一个指向其他线程栈的指针，那么它就可以读写这个栈的任何部分。我们的示例程序在第 26 行展示了这一点，其中对等线程直接通过全局变量 ptr 引用主线程的栈的内容。

13.4.2 将变量映射到存储器

多线程的 C 程序中的变量根据它们的存储类型被映射到虚拟存储器。

- 全局变量。全局变量是定义在函数之外的变量。在运行时，虚拟存储器的读/写区域只包含每个全局变量的一个实例。例如，第 5 行声明的全局变量 ptr 在虚拟存储器的读/写区域中有一个运行时实例。当一个变量只有一个实例时，我们只用变量名——在这里就是 ptr——来表示这个实例。
- 本地自动变量。本地自动变量就是定义在函数内部但是没有 static 属性的变量。在运行时，每个线程的栈都包含它自己的所有本地自动变量的实例。即使多个线程执行同一个线程例程时，也是如此。例如，有一个本地变量 tid 的实例，它保存在主线程的栈中。我们用 tid.m 来表示这个实例。再来看一个例子，本地变量 myid 有两个实例，一个在对等线程 0 的栈内，另一个在对等线程 1 的栈内。我们将这两个实例分别表示为 myid.p0 和 myid.p1。
- 本地静态变量。本地静态变量是定义在函数内部并有 static 属性的变量。和全局变量一样，虚拟存储器的读/写区域只包含在程序中声明的每个本地静态变量的一个实例。例如，即使我们示例程序中的每个对等线程都在第 25 行声明了 cnt，在运行时，虚拟存储器的读/写区域中也只有一个 cnt 的实例。每个对等线程都读和写这个实例。

13.4.3 共享变量

我们说一个变量 v 是共享的，当且仅当它的一个实例被一个以上的线程引用。例如，我们示例程序中的变量 cnt 就是共享的，因为它只有一个运行时实例，并且这个实例被两个对等线程引用。在另一方面，myid 不是共享的，因为它的两个实例中每一个都只被一个线程引用。然而，认识到像 msgs 这样的本地自动变量也能被共享是很重要的。

练习题 13.6

A. 利用 13.4 节中的分析，为图 13.15 中的示例程序在下表的每个条目中填写“是”或者“否”。在第一列中，符号 v.t 表示变量 v 的一个实例，它驻留在线程 t 的本地栈中，其中 t 要么是 m（主线程），要么是 p0（对等线程 0）或者 p1（对等线程 1）。

变量实例	主线程引用的？	对等线程 0 引用的？	对等线程 1 引用的？
ptr			
cnt			
i.m			

（续表）

变量实例	主线程引用的？	对等线程 0 引用的？	对等线程 1 引用的？
msgs.m			
myid.p0			
myid.p1			

B. 根据 A 部分的分析，变量 ptr、cnt、i、msgs 和 myid 哪些是共享的？

13.5 用信号量同步线程

共享变量是十分方便，但是它们也引入了同步错误（synchronization error）的可能性。考虑图 13.16 中的程序 badcnt.c，它创建了两个线程，每个线程都对共享计数变量 cnt 加 1。

code/conc/badcnt.c

```
1   #include "csapp.h"
2
3   #define NITERS 100000000
4   void *count(void *arg);
5
6   /* shared counter variable */
7   unsigned int cnt = 0;
8
9   int main()
10  {
11      pthread_t tid1, tid2;
12
13      Pthread_create(&tid1, NULL, count, NULL);
14      Pthread_create(&tid2, NULL, count, NULL);
15      Pthread_join(tid1, NULL);
16      Pthread_join(tid2, NULL);
17
18      if (cnt != (unsigned)NITERS*2)
19          printf("BOOM! cnt=%d\n", cnt);
20      else
21          printf("OK cnt=%d\n", cnt);
22      exit(0);
23  }
24
25  /* thread routine */
26  void *count(void *arg)
27  {
28      int i;
29      for (i = 0; i < NITERS; i++)
30          cnt++;
```

```
31        return NULL;
32    }
```

code/conc/badcnt.c

图 13.16 badcnt.c：一个不正确同步化的计数器程序

因为每个线程都对计数器增加了 NITERS 次，我们可能会预计它的最终值是 2*NITERS。然而，当我们在我们的系统上运行 badcnt.c 时，我们不仅得到错误的答案，而且每次得到的答案都还不相同！

```
unix> ./badcnt
BOOM! ctr=198841183
unix> ./badcnt
BOOM! ctr=198261801
unix> ./badcnt
BOOM! ctr=198269672
```

那么哪里出错了呢？为了清晰地理解这个问题，我们需要研究计数值循环的汇编代码，如图 13.17 所示。我们发现，将线程 i 的循环代码分解成五个部分是很有帮助的：

- H_i：在循环头部的指令块。
- L_i：加载共享变量 cnt 到寄存器%eax$_i$ 的指令，这里%eax$_i$ 表示线程 i 中的寄存器%eax 的值。
- U_i：更新（增加）%eax$_i$ 的指令。
- S_i：将%eax$_i$ 的更新值存回到共享变量 cnt 的指令。
- T_i：循环尾部的指令块。

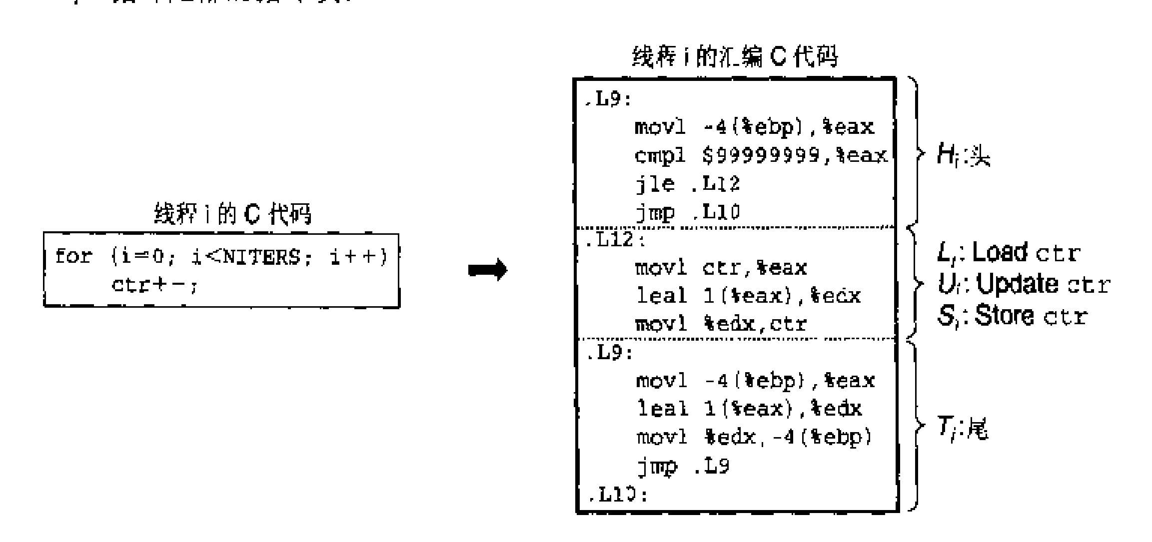


图 13.17 badcnt.c 中计数器循环的 IA32 汇编代码

注意头和尾只操作本地栈变量，而 L_i、U_i 和 S_i 操作共享计数器变量的内容。

当 badcnt.c 中的两个对等线程在一个单处理器上并发运行时，机器指令以某种顺序一个接一个地完成。因此，每个并发执行定义了两个线程中的指令的某种全序（或者交互）。不幸地是，这些顺序中的一些将会产生正确结果，但是其他的则不会。

这里有个关键点：一般而言，你没有办法预测操作系统是否将为你的线程选择一个正确的顺

序。例如，图 13.18（a）展示了一个正确的指令顺序的分步操作。在每个线程更新了共享变量 cnt 之后，它在存储器中的值就是 2，这正是期望的值。另一方面，图 13.18（b）的顺序产生一个不正确的 cnt 的值。会发生这样的问题是因为，线程 2 在第 5 步加载 cnt，是在第 2 步线程 1 加载 cnt 之后，而在第 6 步线程 1 存储它的更新值之前。因此，每个线程结束时都会存储一个值为 1 的更新后的计数器值。

步骤	线程	指令	%eax$_1$	%eax$_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

（a）正确的顺序

步骤	线程	指令	%eax$_1$	%eax$_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

（b）不正确的顺序

图 13.18 badcnt.c 中第一次循环反复中的指令顺序

我们能够借助于一种叫做进度图（progress graph）的设备来阐明这些正确的和不正确的指令顺序的概念，这个图我们将在下一节介绍。

练习题 13.7

完成下表中 badcnt.c 的指令顺序：

步骤	线程	指令	$\%eax_1$	$\%eax_2$	cnt
1	1	H_1	—	—	0
2	1	L_1			
3	2	H_2			
4	2	L_2			
5	2	U_2			
6	2	S_2			
7	1	U_1			
8	1	S_1			
9	1	T_1			
10	2	T_2			

这种顺序会产生一个正确的 cnt 值吗？

13.5.1 进度图

一个进度图（progress graph）将 n 个并发线程的执行模型化为一条 n 维笛卡儿空间中的轨线。每条轴 k 对应于线程 k 的进度。每个点（$I_1,I_2,\cdots,I_n$）代表线程 k（k=1,…,n）已经完成了指令 I_k 这一状态。图的原点对应于没有任何线程完成一条指令的初始状态。

图 13.19 展示了 badcnt.c 程序第一次循环迭代的二维进度图。水平轴对应于线程 1，垂直轴对应于线程 2。点（L_1，S_2）对应于线程 1 完成了 L_1 而线程 2 完成了 S_2 的状态。

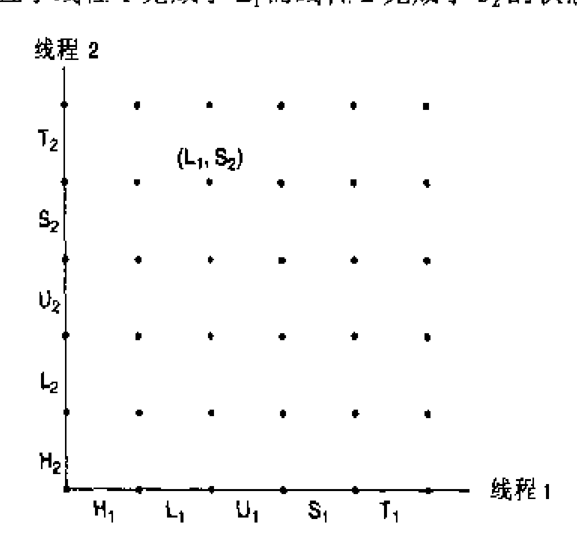


图 13.19 badcnt.c 第一次循环迭代的进度图

一个进度图将指令执行模型化为一个从一种状态到另一种状态的转换（transition）。一个转换被表示为一条从一点到相邻点的有向边。合法的转换是向右（线程 1 中的一条指令完成）或者向上（线程 2 中的一条指令完成）的。两个指令不能在同一时刻完成——对角线转换是不允许的。程序决不会反向运行，所以向下或者向左移动的转换也是不合法的。

一个程序的执行历史被模型化为状态空间中的一条轨线。图 13.20 展示了下面指令顺序对应的轨线：

H_1，L_1，U_1，H_2，L_2，S_1，T_1，U_2，S_2，T_2

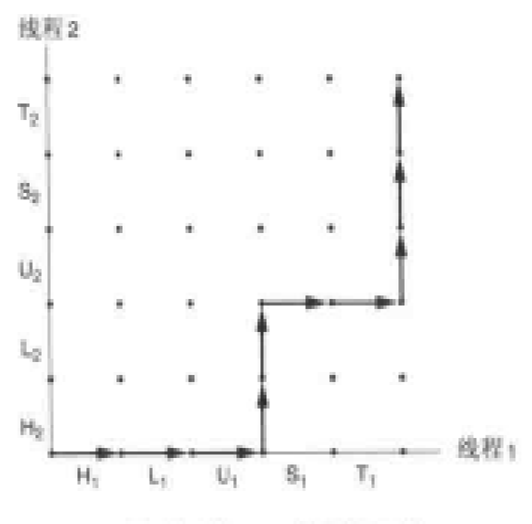


图 13.20 一个轨线示例

对于线程 i，操作共享变量 cnt 内容的指令（L_i，U_i，S_i）构成了一个（关于共享变量 cnt 的）临界区（critical section），这个临界区不应该和其他进程的临界区交替执行。两个临界区的交集形成的状态空间称为不安全区（unsafe region）。图 12.31 展示了变量 cnt 的不安全区。注意，不安全区和与它交界的状态相毗邻，但并不包括这些状态。例如，状态（H_1，H_2）和（S_1，U_2）毗邻不安全区，但是它们并不是不安全区的一部分。

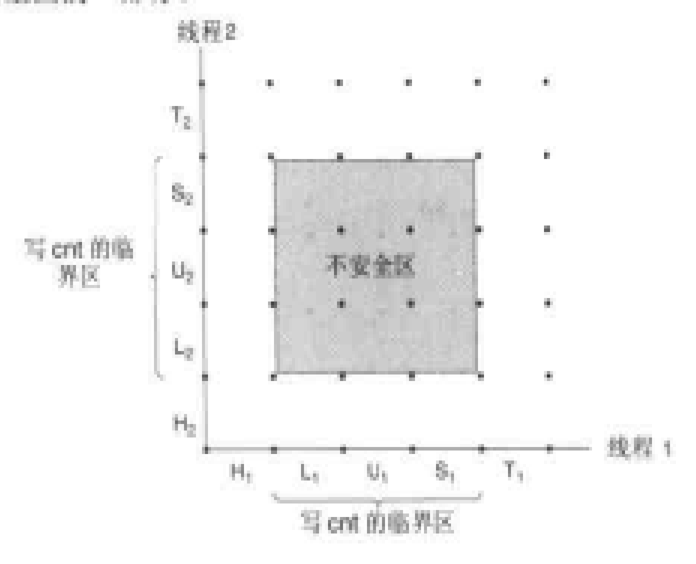


图 13.21 临界区和不安全区

环绕不安全区的轨线叫做**安全轨线**（safe trajectory）。相反，接触任何不安全区的轨线就叫做**不安全轨线**（unsafe trajectory）。图 13.22 给出了我们的示例程序 badcnt.cd 的状态空间中的安全和不安全轨线。上面的轨线环绕不安全区域的左边和上边，所以是安全的。下面的轨线穿越不安全区，因此是不安全的。

任何安全轨线都将正确地更新共享计数器。为了保证我们的多线程程序示例的正确执行——和实际上任何共享全局数据结构的并发程序的正确执行——我们必须以某种方式同步线程，使它们总是有一条安全轨线。

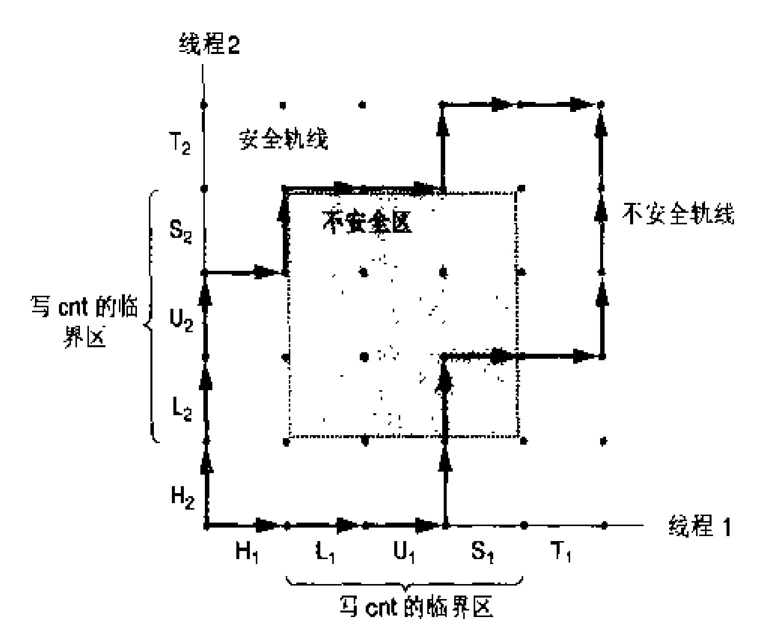


图 13.22 安全和不安全轨线

13.5.2 利用信号量访问共享变量

Edsger Dijkstra，理解和阐明并发编程领域的先锋人物，提出了一种经典的解决同步不同执行线程问题的方法，这种方法是基于一种叫做信号量（semaphore）的特殊类型变量的。信号量 s 是具有非负整数值的全局变量，只能由两种特殊的操作来处理，这两种操作称为 P 和 V。

- $P(s)$：如果 s 是非零的，那么 P 将 s 减 1，并且立即返回。如果 s 为零，那么就挂起进程，直到 s 变为非零，并且该进程被一个 V 操作重启。在重启之后，P 操作将 s 减 1，并将控制返回给调用者。
- $V(s)$：V 操作将 s 加 1。如果有任何进程阻塞在 P 操作等待 s 变成非零，那么 V 操作会重启这些进程中的一个，然后该进程将 s 减 1，完成它的 P 操作。

P 中的测试和减 1 操作是不可分割的，也就是说，一旦预测 s 变为非零，就会将 s 减 1，不能有中断。V 中的加 1 操作也是不可分割的，也就是加载、加 1 和存储信号量的过程中没有中断。

旁注：名字 P 和 V 的起源

Edsger Dijkstra 出生于荷兰，名字 P 和 V 来源于荷兰语单词 *Proberen*（测试）和 *Verhogen*（增加）。

P 和 V 的定义确保了一个运行程序绝不可能进入这样一种状态，也就是一个正确初始化了的信号量有一个负值。这个属性称为*信号量不变性*（semaphore invariant），为控制并发程序的轨线而避免不安全区提供了强有力的工具。

基本的思想是将每个共享变量（或者相关共享变量集合）与一个信号量 s（初始为 1）联系起来，然后用 $P(s)$和 $V(s)$操作将相应的临界区包围起来。以这种方式来保护共享变量的信号量叫做*二进制信号量*（binary semaphore），因为它的值总是 0 或者 1。

图 13.23 中的进度图展示了我们如何利用信号量来正确地同步我们的计数器程序示例。每个状态都标出了该状态中信号量 s 的值。关键概念是这种 P 和 V 操作的结合创建了一组状态，叫做禁止

区（forbidden region），其中 s<0。因为信号量的不变性，没有实际可行的轨线能够包含禁止区中的状态。而且，因为禁止区完全包括了不安全区，所以没有实际可行的轨线能够接触不安全区的任何部分。因此，每条实际可行的轨线都是安全的，而且不管运行时指令顺序是怎样的，程序都会正确地增加计数器值。

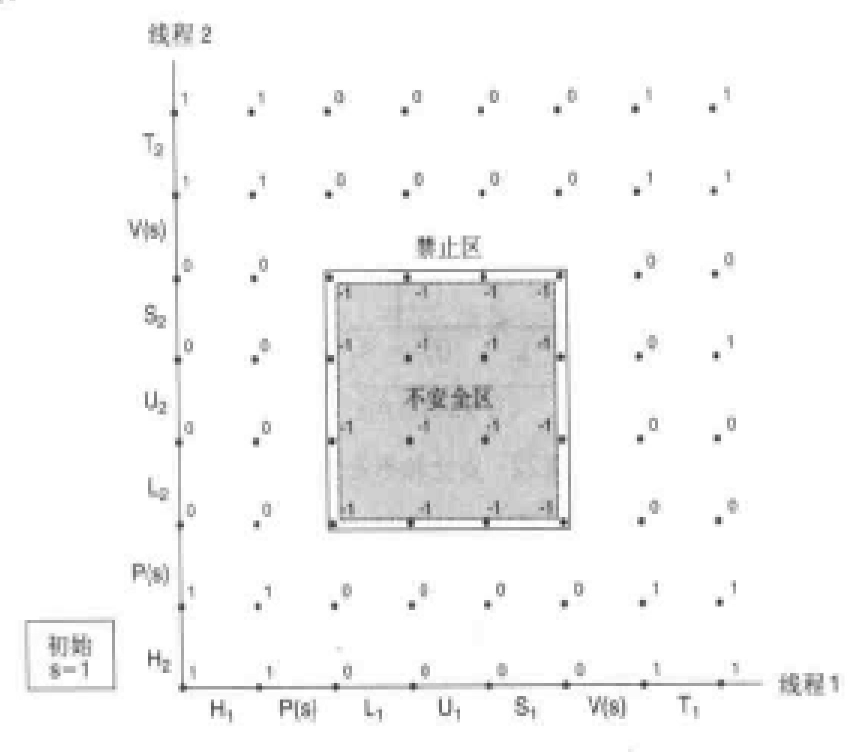


图 13.23 使用信号量的安全共享

s<0 的状态定义了一个包括不安全区的禁止区。

从可操作的意义上来说，由 *P* 和 *V* 操作创建的禁止区使得在任何时间点上，在被包围的临界区中，不可能有多个线程在执行指令。换句话说，信号量操作确保了对临界区的*互斥访问*（mutually exclusive access）。一般现象称为*互斥*（mutual exclusion）。

一点行话：目的是提供互斥的二进制信号量通常叫做*互斥锁*（mutex）。在互斥锁上执行一个 *P* 操作叫做*加锁*。相似地，执行 *V* 操作叫做*解锁*。一个已经对一个互斥锁加锁但还没有解锁的线程被称为*占用*互斥锁。

旁注：进度图的局限性

进度图给了我们一种较好的方法，将在单处理器上的并发程序执行可视化，也帮助我们理解为什么需要同步。然而，它们确实也有局限性，特别是对于在多处理器上的并发执行，一组 CPU/高速缓存对共享相同的存储器（或主存），多处理器的工作方式是进度图不能解释的。特别是，一个多处理器存储系统可以处于一种状态，不对应于进度图中任何轨线。不管如何，结论总是一样的：要同步对你共享变量的访问。

13.5.3 Posix 信号量

Posix 标准定义了许多操作信号量的函数。三个基本的操作是 sem_init、sem_wait（*P* 操作）和 sem_post（*V* 操作）。

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

返回：若成功则为0，若出错则为-1。

一个程序通过调用 sem_init 函数来初始化一个信号量。sem_init 函数将信号量 sem 初始化为 value。每个信号量在使用前必须初始化。针对我们的目的，中间的参数总是零。程序分别通过调用 sem_wait 和 sem_post 函数来执行 *P* 和 *V* 操作。为了简明，我们更喜欢使用下面的 *P* 和 *V* 包装(wrapper)函数：

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

返回：无。

例如，为了正确同步我们的计数器示例，我们可以声明一个叫做 mutex 的信号量：

```
sem_t mutex;
```

接下来，在主例程中，我们将它初始化为一：

```
sem_init (&mutex, 0, 1);
```

最后，我们利用对 mutex 的 *P* 和 *V* 操作包围 cnt 变量，从而保护它：

```
P(&mutex);
Cnt++;
V(&mutex);
```

13.5.4 利用信号量来调度共享资源

我们在前一小节里看到了如何用信号量来提供对共享变量的互斥访问。信号量的另一个重要作用是调度对共享资源的访问。在这种情况中，一个线程用信号量来通知另一个线程，程序状态中的某个条件已经为真了。图 13.24 所示的生产者和消费者模型是一个经典的示例。生产者和消费者线程共享一个有 *n* 个槽的有界缓冲区。



图 13.24 生产者-消费者模型

生产者产生项目（item）并把它们插入到缓冲区中。消费者从缓冲区中取出这些项目并以某种方式使用它们。

生产者线程反复地生成新的项目（item），并把它们插入到缓冲区中。消费者线程不断地从缓冲区中取出这些项目，然后消费它们。模型中也可能有不同的生产者和消费者数量。

因为插入和取出项目都包括更新共享变量，所以我们必须保证对缓冲区的访问是互斥的。但是只保证互斥访问是不够的，我们还需要调度对缓冲区的访问。如果缓冲区是满的（没有空的槽位），那么生产者必须等待直到有一个槽位变为可用。与之相似，如果缓冲区是空的（没有可取用的项目），那么消费者必须等待直到有一个可用的项目。

生产者-消费者的相互作用在现实系统中是很普遍的。例如，在一个多媒体系统中，生产者编码视频帧，而消费者解码并在屏幕上呈现出来。缓冲区的目的是为了减少视频流的抖动（jitter），而这种抖动是由各个帧的编码和解码时与数据相关的差异引起的。缓冲区为生产者提供了一个槽位池，而为消费者提供一个已编码的帧池。另一个常见的示例是图形用户接口的设计。生产者检测到鼠标和键盘事件，并将它们插入到缓冲区中。消费者以某种基于优先级的方式从缓冲区取出这些事件，并画在屏幕上。

在本节中，我们将开发一个简单的包，叫做 Sbvf 用来构造生产者-消费者程序。在下一节里，我们会看到如何用它来构造基于预线程化（prethreading）的一个有趣的并发服务器。Sbvf 操作类型为 sbuf_t 的缓冲区（图 13.25）。项目存放在一个动态分配的 n 项整数数组中。front 和 rear 索引值记录该数组中的第一项和最后一项。三个信号量控制对缓冲区的同步访问。mutex 信号量提供互斥的缓冲区访问。slots 和 items 信号量分别记录空槽位和可用项目的数量。

code/conc/sbuf.h

```
1   typedef struct {
2       int *buf;          /* Buffer array */
3       int n;             /* Maximum number of slots */
4       int front;         /* buf[(front+1)%n] is first item */
5       int rear;          /* buf[rear%n] is last item */
6       sem_t mutex;       /* Protects accesses to buf */
7       sem_t slots;       /* Counts available slots */
8       sem_t items;       /* Counts available items */
9   } sbuf_
```

code/conc/sbuf.h

图 13.25 sbuf_t：生产者-消费者程序的一个共享缓冲区

sbuf_init 函数（图 13.26）为缓冲区分配堆存储器，设置 front 和 rear 表示一个空的缓冲区，并为三个信号量赋初始值。这个函数在调用其他三个函数中的任何一个之前调用一次。

code/conc/sbuf.c

```
1   void sbuf_init(sbuf_t *sp, int n)
2   {
3       sp->buf = Calloc(n, sizeof(int));
4       sp->n = n;                        /* Buffer holds max of n items */
5       sp->front = sp->rear = 0;         /* Empty buffer iff front == rear */
6       Sem_init(&sp->mutex, 0, 1);       /* Binary semaphore for locking */
7       Sem_init(&sp->slots, 0, n);       /* Initially, buf has n empty slots */
8       Sem_init(&sp->items, 0, 0);       /* Initially, buf has zero data items */
9   }
```

code/conc/sbuf.c

图 13.26 sbuf_init：初始化一个共享缓冲区

sbuf_deinit 函数（没有显示出来）是当应用程序使用完缓冲区时，释放缓冲区存储的。sbuf_insert 函数（图 13.27）等待一个可用的槽位、对互斥锁加锁、添加项目、对互斥锁解锁，然后宣布有一个新项目可用。

code/conc/sbuf.c
```
1   void sbuf_insert(sbuf_t *sp, int item)
2   {
3        P(&sp->slots);                        /* Wait for available slot */
4        P(&sp->mutex);                        /* Lock the buffer */
5       sp->buf[(++sp->rear)%(sp->n)] = item;  /* Insert the item */
6       V(&sp->mutex);                         /* Unlock the buffer */
7       V(&sp->items);                         /* Announce available item */
8   }
```
code/conc/sbuf.c

图 13.27 sbuf_inser：在一个共享缓冲区的后部插入一个项目
这个函数一直等待到有一个槽位可用。

sbuf_remove 函数（图 13.28）是与 sbuf_insert 函数对称的。在等待一个可用的缓冲区项目之后、对互斥锁加锁、从缓冲区的前面取出该项目、对互斥锁解锁，然后发信号通知一个新的槽位可供使用。

code/conc/sbuf.c
```
1   int sbuf_remove(sbuf_t *sp)
2   {
3       int item;
4       P(&sp->items);                          /* Wait for available item */
5       P(&sp->mutex);                          /* Lock the buffer */
6       item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
7       V(&sp->mutex);                          /* Unlock the buffer */
8       V(&sp->slots);                          /* Announce available slot */
9       return item;
10  }
```
code/conc/sbuf.c

图 13.28 sbuf_remove：从一个共享缓冲区的前部取出一个项目
这个函数一直等待到有一个项目可用。

旁注：其他同步机制

我们已经向你展示了如何利用信号量来同步线程，主要是因为它们简单、经典，并且有一个清晰的语义模型。但是你应该知道也是存在着其他同步技术。例如，Java 线程是用一种叫做 Java 监控器（Java Monitor）[34]的机制来同步的，它提供了对信号量互斥和调度能力的更高级别的抽象；实际上，监控器可以用信号量来实现。再来看一个例子，Pthreads 接口定义了一组对互斥锁和条件变量的同步操作。Pthreads 互斥锁被用来实现互斥。条件变量用来调度对共享资源的访问，例如在一个生产者-消费者程序中的有界缓冲区。

13.6　综合：基于预线程化的并发服务器

我们已经知道了如何使用信号量来访问共享变量和调度对共享资源的访问。为了帮助你更清晰地理解这些思想，让我们把它们应用到一个基于被称为预线程化（prethreading）技术的并发服务器上。

在图 13.14 中的并发服务器中，我们为每一个新客户端创建了一个新线程。这种方法的缺点是我们为每一个新客户端创建一个新线程，导致不小的代价。一个基于预线程化的服务器通过使用图 13.29 所示的生产者-消费者模型来试图降低这种开销。

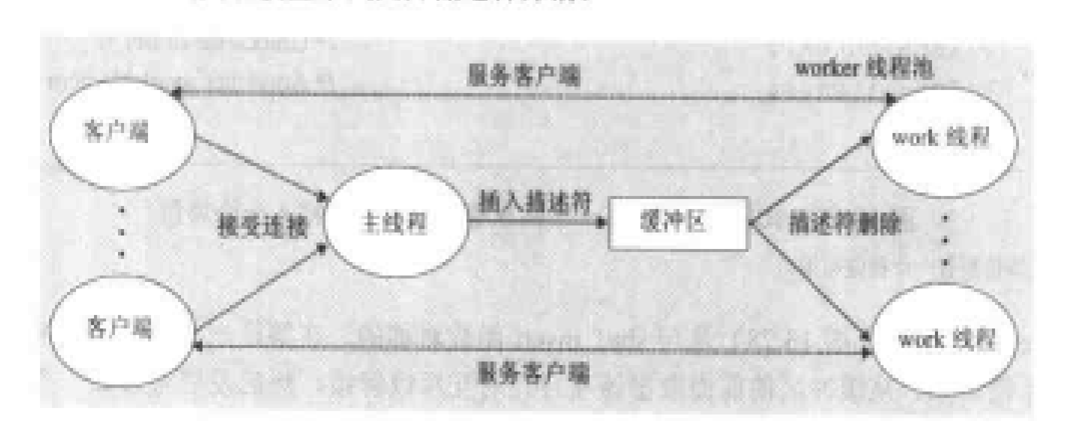


图 13.29　预线程化的并发服务器的组织结构

一组现有的线程不断地删除和处理来自共享缓冲区的连接描述符。

服务器是由一个主线程和一组 worker 线程构成的。主线程不断地接受来自客户端的连接请求，并将得到的连接描述符放在一个共享缓冲区中。每一个 worker 线程反复地从共享缓冲区中取出描述符，为客户端服务，然后等待下一个描述符。

图 13.30 显示了我们怎样用 Sbvf 包来实现一个预线程化的并发 echo 服务器。在初始化了缓冲区 sbuf（第 22 行）后，主线程创建了一组 worker 线程（第 25～26 行）。然后它进入了无限的服务器循环，接受连接请求，并将得到的连接描述符插入到缓冲区 sbuf 中。每个 worker 线程的行为都非常简单。它等待直到它能从缓冲区中取出一个已连接描述符（第 38 行），然后调用 echo_cnt 函数回送客户端的输入。

code/conc/echoservert_pre.c

```
1   #include "csapp.h"
2   #include "sbuf.h"
3   #define NTHREADS 4
4   #define SBUFSIZE 16
5
6   void echo_cnt(int connfd);
7   void *thread(void *vargp);
8
9   sbuf_t sbuf; /* shared buffer of connected descriptors */
10
11  int main(int argc, char **argv)
```

```
12  {
13      int i, listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
14      struct sockaddr_in clientaddr;
15      pthread_t tid;
16
17      if (argc != 2) {
18          fprintf(stderr, "usage: %s <port>\n", argv[0]);
19          exit(0);
20      }
21      port = atoi(argv[1]);
22      sbuf_init(&sbuf, SBUFSIZE);
23      listenfd = Open_listenfd(port);
24
25      for (i = 0; i < NTHREADS; i++) /* Create worker threads */
26          Pthread_create(&tid, NULL, thread, NULL);
27
28      while (1) {
29          connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
30          sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
31      }
32  }
33
34  void *thread(void *vargp)
35  {
36      Pthread_detach(pthread_self());
37      while (1) {
38          int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
39          echo_cnt(connfd);                /* Service client */
40          Close(connfd);
41      }
42  }
```
code/conc/echoservert_pre.c

图 13.30 一个预线程化的并发 echo 服务器

这个服务器使用的是有一个生产者和多个消费者的生产者-消费者模型。

函数 echo_cnt（图 13.31）是图 12.21 中的 echo 函数的一个版本，它在全局变量 byte_cnt 中记录了从所有客户端接收到的累计字节数。

code/conc/echo_cnt.c
```
1   #include "csapp.h"
2
3   static int byte_cnt;  /* byte counter */
4   static sem_t mutex;   /* and the mutex that protects it */
5
6   static void init_echo_cnt(void)
```

```
 7  {
 8      Sem_init(&mutex, 0, 1);
 9      byte_cnt = 0;
10  }
11
12  void echo_cnt(int connfd)
13  {
14      int n;
15      char buf[MAXLINE];
16      rio_t rio;
17      static pthread_once_t once = PTHREAD_ONCE_INIT;
18
19      Pthread_once(&once, init_echo_cnt);
20      Rio_readinitb(&rio, connfd);
21      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
22          P(&mutex);
23          byte_cnt += n;
24          printf("thread %d received %d (%d total) bytes on fd %d\n",
25                 (int) pthread_self(), n, byte_cnt, connfd);
26          V(&mutex);
27          Rio_writen(connfd, buf, n);
28      }
29  }
```

code/conc/echo_cnt.c

图 13.31　echo_cnt：echo 的一个版本，它对从客户端接收的所有字节计数

这是一段值得研究的有趣代码，因为它向你展示了一个被线程例程调用的初始化程序包的一般技术。在我们的情况中，我们需要初始化 byte_cnt 计数器和 mutex 信号量。一个方法是我们为 Sbuf 和 Rio 程序包使用的，它要求主线程显式地调用一个初始化函数。另外一个方法，在此显示的，是当第一次有某个线程调用 echo_cnt 函数时，使用 pthread_once 函数（第 19 行）去调用初始化函数。这个方法的优点是它使程序包的使用更加容易。这种方法的缺点是每一次调用 echo_cnt 都会导致调用 pthread_once 函数，而在很多时候它没有做什么有用的事。

一旦程序包被初始化，echo_cnt 函数会初始化 Rio 带缓冲 I/O 包（第 20 行），然后回送从客户端接收到的每一个文本行。注意，对第 23～24 行中共享变量 byte_cnt 的访问是被 P 和 V 操作保护的。

旁注：基于线程的事件驱动程序

I/O 多路复用不是编写事件驱动程序的唯一方法。例如，你可能已经注意到我们刚才设计的并发的预线程化的服务器确实是一个事件驱动服务器，带有主线程和 worker 线程的简单状态机。主线程有两种状态（“等待连接请求”和“等待可用的缓冲区槽位”）、两个 I/O 事件（“连接请求到达”和“缓冲区槽位变为可用”）和两个转换（“接受连接请求”和“插入缓冲区项目”）。同样，每个 worker 线程有一个状态（“等待可用的缓冲项目”）、一个 I/O 事件（“缓冲区项目变为可用”）、一个转换（“取出缓冲区项目”）。

13.7 其他并发性问题

你可能已经注意到了，一旦我们要求同步访问共享数据，那么事情就变得更加复杂了。迄今为止，我们已经看到了关于互斥和生产者-消费者的同步化技术，但这仅仅是冰山一角。同步化是非常困难的，引出了在普通的顺序程序中不会出现的问题。这一小节是关于你在写并发程序时需要注意的一些问题的概括（决不是全面的概括）。为了更加具体化，我们将以线程的形式描述我们的讨论。不过要记住，这些典型问题是任何类型的并发流操作共享资源时都会出现的。

13.7.1 线程安全

当我们用线程编写程序时，我们必须小心地编写那些具有称为线程安全性（thread safety）属性的函数。一个函数被称为**线程安全的**（thread-safe），当且仅当被多个并发线程反复地调用时，它会一直产生正确的结果。如果一个函数不是线程安全的，我们就说它是**线程不安全的**（thread-unsafe）。我们能够定义出四类（有相交的）线程不安全函数：

第 1 类：不保护共享变量的函数

我们在图 13.16 的 count 函数中就已经遇到了这样的问题，该函数对一个未受保护的全局计数器变量加 1。将这类线程不安全函数变成线程安全的，相对而言比较容易：利用像 *P* 和 *V* 操作这样的同步操作来保护共享的变量。这个方法的优点是在调用程序中不需要做任何修改，缺点是同步操作将减慢程序的执行时间。

第 2 类：保持跨越多个调用的状态的函数

一个伪随机数生成器是这类线程不安全函数的简单例子。请参考图 13.32 中的伪随机数生成器程序包。

code/conc/rand.c

```
1   unsigned int next = 1;
2
3   /* rand - return pseudo-random integer on 0..32767 */
4   int rand(void)
5   {
6       next = next*1103515245 + 12345;
7       return (unsigned int)(next/65536) % 32768;
8   }
9
10  /* srand - set seed for rand() */
11  void srand(unsigned int seed)
12  {
13      next = seed;
14  }
```

code/conc/rand.c

图 13.32 一个线程不安全的伪随机数生成器[40]

rand 函数是线程不安全的，因为当前调用的结果依赖于前次调用的中间结果。当我们调用 srand 为 rand 设置了一个种子后，我们反复地从一个单线程中调用 rand，我们能够预期得到一个可重复的

随机数字序列。然而，如果多线程调用 rand 函数，这种假设就不再成立了。

使得rand函数为线程安全的惟一方式是重写它，使得它不再使用任何静态数据，取而代之地依靠调用者在参数中传递状态信息。这样做的缺点是，程序员现在还要被迫修改调用程序中的代码。在一个大的程序中，可能有成百上千个不同的调用位置，做这样的修改将是非常麻烦的，而且还容易出错。

第 3 类：返回指向静态变量的指针的函数

某些函数（例如 gethostbyname）将计算结果放在静态结构中，并返回一个指向这个结构的指针。如果我们从并发线程中调用这些函数，那么将可能发生灾难，因为正在被一个线程使用的结果会被另一个线程悄悄地覆盖了。

有两种方法来处理这类线程不安全函数。一种选择是重写函数，使得调用者传递存放结果的结构的地址。这就消除了所有共享数据，但是它要求程序员还要改写调用者中的代码。

如果线程不安全函数是难以修改或不可能修改的（例如，它是从一个库中链接过来的），那么另外一种选择就是使用我们称为 lock-and-copy（加锁-拷贝）的技术。这个概念将线程不安全函数与互斥锁联系了起来。在每一个调用位置，对互斥锁加锁，调用线程不安全函数，动态地为结果分配存储器，拷贝函数返回的结果到这个存储器位置，然后对互斥锁解锁。一个吸引人的变化是定义了一个线程安全的包装（wrapper）函数，它执行 lock-and-copy，然后通过调用这个包装函数来取代所有对线程不安全函数的调用。例如，图 13.33 给出了一个 gethostbyname 的线程安全的版本，利用的就是 lock-and-copy 技术。

第 4 类：调用线程不安全函数的函数

如果函数 f 调用线程不安全函数 g，那么 f 就是线程不安全的吗？不一定。如果 g 是第 2 类函数，即依赖于跨越多次调用的状态，那么 f 也是线程不安全的，而且除了重写 g 以外，没有什么办法。然而，如果 g 是第 1 类或者第 3 类函数，那么只要你用一个互斥锁保护调用位置和任何得到的共享数据，f 可能仍然是线程安全的。在图 13.33 中我们看到了一个这种情况很好的示例，其中我们使用 lock-and-copy 编写了一个线程安全函数，它调用了一个线程不安全的函数。

code/conc/gethostbyname_ts.c

```
1   struct hostent *gethostbyname_ts(char *hostname)
2   {
3       struct hostent *sharedp, *unsharedp;
4
5       unsharedp = Malloc(sizeof(struct hostent));
6       P(&mutex);
7       sharedp = gethostbyname(hostname);
8       *unsharedp = *sharedp; /* copy shared struct to private struct */
9       V(&mutex);
10      return unsharedp;
11  }
```

code/conc/gethostbyname_ts.c

图 13.33 gethostbyname_ts：gethostbyname 的一个线程安全的包装函数

使用 lock-and-copy 技术调用一个第 2 类线程不安全函数。

13.7.2 可重入性

有一类重要的线程安全函数，叫做可重入函数（reentrant function），其特点在于它们具有这样一种属性：当它们被多个线程调用时，不会引用任何共享数据。

尽管线程安全和可重入有时会（不正确地）被用做同义词，但是它们之间还是有清晰的技术差别，值得留意。图 13.34 展示了可重入函数、线程安全函数和线程不安全函数之间的集合关系。所有函数的集合被划分成不相交的线程安全和线程不安全函数集合。可重入函数集合是线程安全函数的一个真子集。

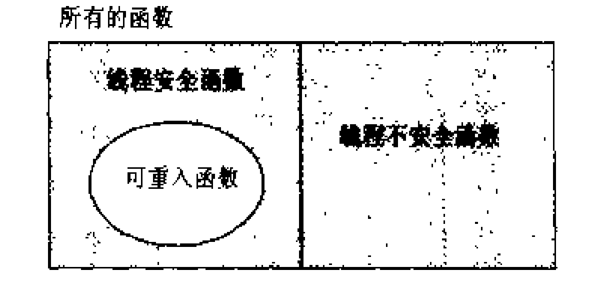


图 13.34 可重入函数、线程安全函数和线程不安全函数之间的集合关系

可重入函数通常要比不可重入的线程安全的函数高效一些，因为它们不需要同步操作。更进一步来说，将第 2 类线程不安全函数转化为线程安全函数的惟一方法就是重写它，使之变为可重入的。例如，图 13.35 展示了图 13.32 中 rand 函数的一个可重入的版本。关键思路是我们用一个调用者传递进来的指针取代了静态的 next 变量。

code/conc/rand_r.c

```
1   /* rand_r - a reentrant pseudo-random integer on 0..32767 */
2   int rand_r(unsigned int *nextp)
3   {
4       *nextp = *nextp * 1103515245 + 12345;
5       return (unsigned int)(*nextp / 65536) % 32768;
6   }
```

code/conc/rand_r.c

图 13.35 rand_r：图 13.32 中的 rand 函数的可重入版本

检查某个函数的代码并先验地断定它是可重入的，这可能吗？不幸地是，不一定能这样。如果所有的函数参数都是传值传递的（也就是，没有指针），并且所有的数据引用都是本地的自动栈变量（也就是，没有引用静态或全局变量），那么函数就是显式可重入的（explicitly reentrant），也就是说，无论它是被如何调用的，我们都可以断言它是可重入的。

然而，如果把我们的假设放宽松一点，允许显式可重入函数中一些参数是引用传递的（也就是说，我们允许它们传递指针），那么我们就得到了一个隐式可重入的（implicitly reentrant）函数，也就是说，在调用线程小心地传递指向非共享数据的指针时，它才是可重入的。例如，图 13.35 中的 rand_r 函数就是隐式可重入的。

我们总是使用术语可重入（reentrant）来包括显式可重入函数和隐式可重入函数。然而，认识到可重入性有时同时是调用者和被调用者的属性，并不只是被调用者单独的属性，是非常重要的。

练习题 13.8

图 13.33 中的 gethostbyname_ts 函数是线程安全的，但不是可重入的。请解释说明。

13.7.3 在多线程程序中使用已存在的库函数

大多数 Unix 函数和定义在标准 C 库中的函数（例如 malloc、free、realloc、printf 和 scanf）都是线程安全的，只有一小部分是例外。图 13.36 列出了常见的例外。（参考[81]可以得到一个完整的列表。）

线程不安全函数	线程不安全类	Unix 线程安全版本
rand	2	rand_r
strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(无)
localtime	3	localtime_r

图 13.36 常见的线程不安全库函数

asctime、ctime 和 localtime 函数是在不同时间和数据格式间相互来回转换时经常使用的函数。gethostbyname、gethostbyaddr 和 inet_ntoa 函数是我们在第 12 章中遇到过的、经常使用的网络编程函数。strtok 函数是一个过时了的（也就是不再鼓励使用的）用来分析字符串的函数。

除了 rand 和 strtok 以外，所有这些线程不安全函数都是第 3 类的，它们返回一个指向静态变量的指针。如果我们需要在一个多线程程序中调用这些函数中的某一个，最简单的方法是 lock-and-copy。lock-and-copy 的缺点是额外的同步降低了程序的速度。更进一步，这种方法对像 rand 这样依赖跨越调用的静态状态的第 2 类函数并不有效。因此，Unix 系统提供大多数线程不安全函数的可重入版本。可重入版本的名字总是以 “_r” 后缀结尾。例如，gethostbyname 的可重入版本就叫做 gethostbyname_r。不幸的是，关于 Unix 的可重入函数的文档很差，并且在不同的 Unix 系统上有不同的接口。因为这个原因，我们建议避免使用它们。

13.7.4 竞争

当一个程序的正确性依赖于一个线程要在另一个线程到达 y 点之前到达它的控制流中的 x 点时，就会发生竞争（race）。通常发生竞争是因为程序员假定线程将按照某种特殊的轨线穿过执行状态空间，而忘记了另一条准则规定：多线程程序必须对任何可行的轨线都正确工作。

例子是理解竞争本质的最简单的方法。让我们来看看图 13.37 中的简单程序。主线程创建了四个对等线程，并传递一个指向一个惟一的整数 ID 的指针到每个线程。每个对等线程拷贝它的参数中传递的 ID 到一个局部变量中（第 21 行），然后输出一个包含这个 ID 的信息。

code/conc/race.c

```
1   #include "csapp.h"
2   #define N 4
3
4   void *thread(void *vargp);
5
6   int main()
7   {
8       pthread_t tid[N];
9       int i;
10
11      for (i = 0; i < N; i++)
12          Pthread_create(&tid[i], NULL, thread, &i);
13      for (i = 0; i < N; i++)
14          Pthread_join(tid[i], NULL);
15      exit(0);
16  }
17
18  /* thread routine */
19  void *thread(void *vargp)
20  {
21      int myid = *((int *)vargp);
22      printf("Hello from thread %d\n", myid);
23      return NULL;
24  }
```

code/conc/race.c

图 13.37 一个带竞争的程序

它看上去足够简单，但是当我们在系统上运行这个程序时，我们得到以下不正确的结果：

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

问题是由每个对等线程和主线程之间的竞争引起的。你能发现这个竞争吗？下面是发生的情况：当主线程在第 12 行创建了一个对等线程，它传递了一个指向本地栈变量 i 的指针。在此时，竞争出现在下一次在第 12 行调用 pthread_create 和第 21 行参数的间接引用和赋值之间。如果对等线程在主线程执行第 12 行之前就执行了第 21 行，那么 myid 变量就得到正确的 ID。否则，它就包含的是其他线程的 ID。令人惊慌的是，我们是否得到正确的答案依赖于内核是如何调度线程的执行的。在我

们的系统中它失败了，但是在其他系统中，它可能就能正确工作，让程序员"幸福地"察觉不到程序的严重错误。

为了消除竞争，我们可以动态地为每个整数 ID 分配一个独立的块，并且传递给线程例程一个指向这个块的指针，如图 13.38 所示（第 12~14 行）。请注意线程例程必须释放这些块以避免存储器泄漏。

当我们在系统上运行这个程序时，我们现在得到了正确的结果：

```
unix> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

练习题 13.9

在图 13.38 中，我们可能想要在主线程中的第 15 行后立即释放已分配的存储器块，而不是在对等线程中释放它。但是这会是个坏注意。为什么？

练习题 13.10

A. 在图 13.38 中，我们通过为每个整数 ID 分配一个独立的块来消除竞争。给出一个不调用 malloc 或者 free 函数的不同的方法。

B. 这种方法的利弊是什么？

code/conc/norace.c

```
1   #include "csapp.h"
2   #define N 4
3
4   void *thread(void *vargp);
5
6   int main()
7   {
8       pthread_t tid[N];
9       int i, *ptr;
10
11      for (i = 0; i < N; i++) {
12          ptr = Malloc(sizeof(int));
13          *ptr = i;
14          Pthread_create(&tid[i], NULL, thread, ptr);
15      }
16      for (i = 0; i < N; i++)
```

```
17          Pthread_join(tid[i], NULL);
18      exit(0);
19  }
20
21  /* thread routine */
22  void *thread(void *vargp)
23  {
24      int myid = *((int *)vargp);
25      Free(vargp);
26      printf("Hello from thread %d\n", myid);
27      return NULL;
28  }
```
code/conc/norace.c

图 13.38 图 13.37 中程序的一个没有竞争的正确版本

13.7.5 死锁

信号量引入了一种潜在的令人厌恶的运行时错误，叫做死锁（deadlock），它指的是一组线程被阻塞了，等待一个永远也不会为真的条件。进度图对于理解死锁是一个无价的工具。例如，图 13.39 展示了一对用两个信号量来实现互斥的线程的进程图。从这幅图中，我们能够得到一些关于死锁的重要知识：

- 程序员使用 *P* 和 *V* 操作顺序不当，以至两个信号量的禁止区域（forbidden region）重叠。如果某个执行轨线偶然到达了死锁状态 *d*，那么就不可能有进一步的进展了，因为重叠的禁止区域阻塞了每个合法方向上的进度。换句话说，程序死锁是因为每个线程都在等待其他线程执行一个根本不可能发生的 *V* 操作。
- 重叠的禁止区域引起了一组称为死锁区域（deadlock region）的状态。如果一个轨线偶然到达了一个死锁区域中的状态，那么死锁就是不可避免的了。轨线可以进入死锁区域，但是它们不可能离开。
- 死锁是一个相当困难的问题，因为它不总是可预测的。一些幸运的执行轨线将绕开死锁区域，而其他的将会陷入这个区域。图 13.39 展示了每种情况的一个示例。对于程序员来说，这其中隐含的着实令人惊慌。你可以 1000 次运行一个程序不出任何问题，但是下一次它就有可能会死锁。或者程序在一台机器上可能运行得很好，但是在另外的机器上就会死锁。最糟糕的是，错误常常是不可重复的，因为不同的执行有不同的轨线。

程序死锁有很多原因，要避免死锁一般而言是很困难的。然而，当使用二进制信号量来实现互斥时，如图 13.39 所示，你可以应用下面的简单而有效的规则来避免死锁：

互斥锁加锁顺序规则：如果对于程序中每对互斥锁（*s*，*t*），每个既包含 s 也包含 t 的线程都按照相同的顺序同时对它们加锁，那么这个程序就是无死锁的。

例如，我们可以通过这样的方法来解决图 13.39 中的死锁问题：在每个线程中先对 s 加锁，然

后再对 t 加锁。图 13.40 展示了得到的进度图。

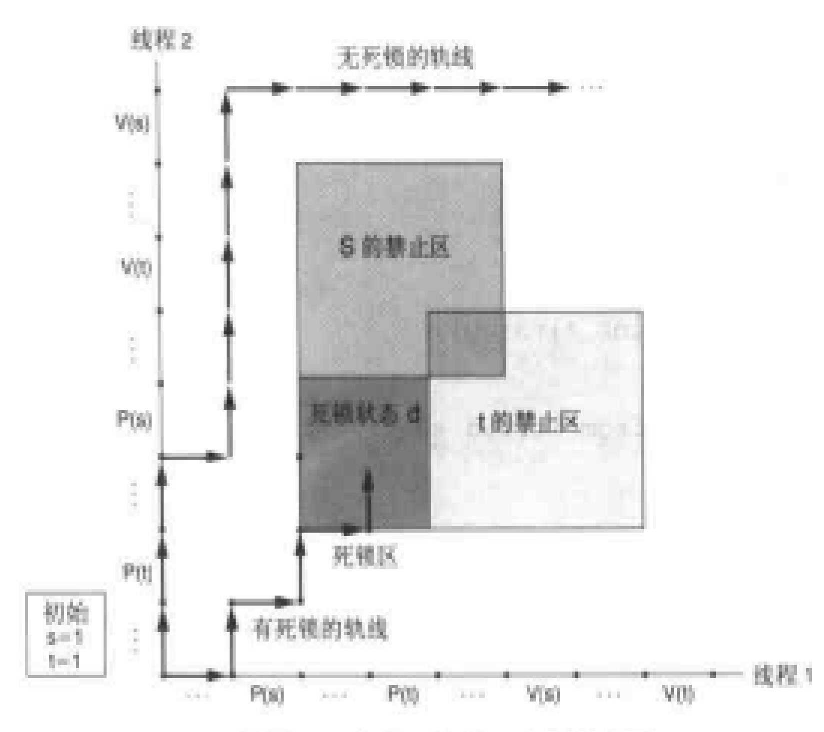


图 13.39 一个会死锁的程序的进度图

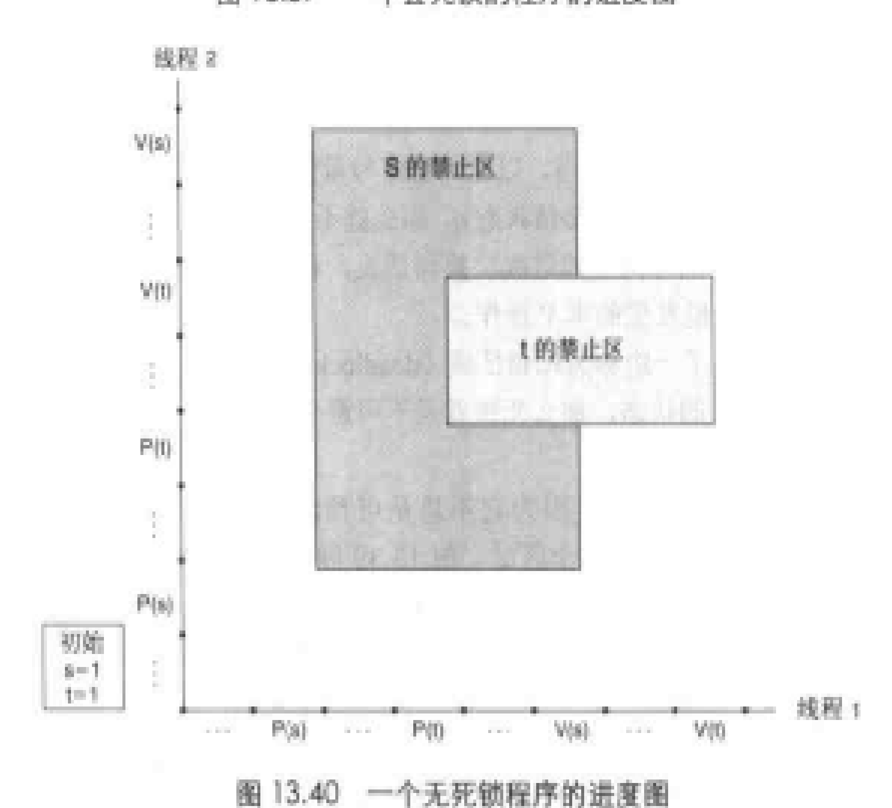


图 13.40 一个无死锁程序的进度图

练习题 13.11

思考下面的程序，它试图使用一对信号量来实现互斥。

初始时：s=1，t=0。

线程 1;　　线程 2;
P(s);　　P(s);
V(s);　　V(s);
P(t);　　P(t);
V(t);　　V(t);

A. 画出这个程序的进度图。

B. 它总是会死锁吗？

C. 如果是，那么对初始信号量的值做什么简单的改变就能消除这种潜在的死锁呢？

D. 画出得到的无死锁程序的进度图。

13.8 小结

一个并发程序是由在时间上重叠的一组逻辑流组成的。在这一章中，我们学习了三种不同的构建并发程序的机制：进程、I/O 多路复用和线程。我们以一个并发网络服务器作为贯穿全章的应用程序。

进程是由内核自动调度的，而且因为它们有各自独立的虚拟地址空间，所以要实现共享数据，它们需要显式的 IPC 机制。事件驱动程序创建它们自己的并发逻辑流，这些逻辑流被模型化为状态机，用 I/O 多路复用来显式地调度这些流。因为程序运行在一个单一进程中，所以在流之间共享数据速度很快而且很容易。线程是这些方法的综合。同基于进程的流一样，线程是由内核自动调度的。同基于 I/O 多路复用的流一样，线程是运行在一个单一进程的上下文中的，因此可以快速而方便地共享数据。

无论哪种并发机制，同步对共享数据的并发访问都是一个困难的问题。提出对信号量的 P 和 V 操作就是为了帮助解决这个问题。信号量操作可以用来提供对共享数据的互斥访问，也对诸如生产者-消费者程序中共享缓冲区这样的资源访问进行调度。一个并发预线程化的 echo 服务器提供了这两种信号量使用场景的很好的例子。

并发性也引入了其他一些困难的问题。被线程调用的函数必须具有一种称为线程安全的属性。我们定义了四类线程不安全函数，以及一些将它们变为线程安全的建议。可重入函数是线程安全函数的一个真子集，它不访问任何共享数据。可重入函数通常比不可重入函数更为有效，因为它们不需要任何同步原语。竞争和死锁是并发程序中出现的另一些困难的问题。当程序员错误地假设逻辑流该如何调度时，就会发生竞争。当一个流等待一个永远不会发生的事件时，就会产生死锁。

参考文献说明

信号量操作是 Dijkstra 提出的[24]。进度图的概念是 Coffman[16]提出的，后来由 Carson 和 Reynolds[10]正式化的。Butenhof 的书[9]对 Posix 线程接口有全面的描述。Birrell[4]的文章对线程编程以及线程编程中容易遇到的问题做了很好的介绍。Pugh 描述了 Java 线程通过存储器进行交互的方式的缺陷，并提出了替代的存储器模型[61]。

家庭作业

13.12 ◆

编写一个 hello.c（图 13.13）的版本，使得它创建和回收 n 个可结合的（joinable）对等线程，其中 n 是一个命令行参数。

13.13 ◆

A. 图 13.41 中的程序有一个 bug。要求线程睡眠一秒，然后输出一个字符串。然而，当在我们的系统上运行它时，却没有任何输出。为什么？

B. 你可以通过用两个不同的 Pthreads 函数调用中的一个替代第 9 行中的 exit 函数，来改正这个错误。选哪一个呢？

code/conc/hellobug.c

```
1   #include "csapp.h"
2   void *thread(void *vargp);
3
4   int main()
5   {
6       pthread_t tid;
7
8       Pthread_create(&tid, NULL, thread, NULL);
9       exit(0);
10  }
11
12  /* thread routine */
13  void *thread(void *vargp)
14  {
15      Sleep(1);
16      printf("Hello, world!\n");
17      return NULL;
18  }
```

code/conc/hellobug.c

图 13.41 习题 13.13 的有 bug 的程序

13.14 ◆◆

检查一下你对 select 函数的理解，请修改图 13.6 中的服务器，使得它每次在主服务器循环中最多只回送一个文本行。

13.15 ◆◆

图 13.8 中的事件驱动并发 echo 服务器是有缺陷的，因为一个恶意的客户端能够通过发送部分的文本行，使服务器拒绝为其他客户端服务。编写一个改进的服务器版本，使之能够非阻塞地处理

这些部分文本行。

13.16 ◆

R_{IO} I/O 包中的函数（11.4 节）都是线程安全的。它们也都是可重入函数吗？

13.17 ◆

在图 13.30 中的预线程化的并发 echo 服务器中，每个线程都调用 echo_cnt 函数（图 13.13）。echo_cnt 是线程安全的吗？它是可重入的吗？为什么是或为什么不是呢？

13.18 ◆◆

一些网络编程的文献建议用以下的方法来读和写套接字：和客户端交互之前，在同一个打开的已连接套接字描述符上，打开两个标准 I/O 流，一个用来读，一个用来写：

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd,"r");
fpout = fdopen(sockfd,"w");
```

当服务器完成和客户端的交互之后，像下面这样关闭两个流：

```
fclose(fpin);
fclose(fpout);
```

然而，如果你试图在基于线程的并发服务器上尝试这种方式，你将制造一个致命的竞争条件。请解释。

13.19 ◆

在图 13.40 中，将两个 V 操作的顺序交换，对程序死锁是否有影响？通过画出四种可能情况的进度图来证明你的答案：

情况 1		情况 2		情况 3		情况 4	
线程 1	线程 2	线程 1	线程 2	线程 1	线程 2	线程 1	线程 2
P(s)	P(s)	P(s)	P(s)	P(s)	P(s)	P(s)	P(s)
P(t)	P(t)	P(t)	P(t)	P(t)	P(t)	P(t)	P(t)
V(s)	V(s)	V(s)	V(t)	V(t)	V(s)	V(t)	V(t)
V(t)	V(t)	V(t)	V(s)	V(s)	V(t)	V(s)	V(s)

13.20 ◆

下面的程序会死锁吗？为什么？

```
Initially:a = 1, b = 1, c = 1.

Thread 1:   Thread 2:
```

```
P(a);        P(c);
P(b);        P(b);
V(b);        V(b)
P(c);        V(c);
V(c);
V(a);
```

13.21 ◆

考虑下面这个会死锁的程序段。

```
Initially:a = 1, b = 1, c = 1.

Thread 1:       Thread 2:       Thread 3
P(a);           P(c);           P(c);
P(b);           P(b);           V(c);
V(b);           V(b);           P(b);
P(c);           V(c);           P(a);
V(c);           P(a);           V(a);
V(a);           V(a);           V(b);
```

A. 列出每个线程同时保持的一对互斥锁。

B. 如果 a<b<c，那么哪个线程违背了互斥锁加锁顺序规则？

C. 对于这些线程，指出一个新的保证不会发生死锁的加锁顺序。

13.22 ◆◆◆

实现标准的 I/O 函数 fgets 的一个版本，叫做 tfgets, 假如它在 5 秒之内没有从标准输入上接收到一个输入行，那么就超时，并返回一个 NULL 指针。你的函数应该实现在一个叫做 tfgets-select.c 的包中，使用进程、信号和非本地跳转。它不应该使用 Unix 的 alarm 函数。使用图 13.42 中的驱动程序测试你的结果。

13.23 ◆◆◆

使用 select 函数来实现练习题 13.22 中 tfgets 函数的一个版本，你的函数应该在一个叫做 tfgets-select.c 的包中实现。用练习题 13.22 中的驱动程序测试你的结果。你可以假定标准输入被赋值为描述符零。

13.24 ◆◆◆

实现练习题 13.22 中 tfgets 函数的一个多线程版本。你的函数应该在一个叫做 tfgets-select.c 的包中实现，用练习题 13.22 中的驱动程序测试你的结果。

13.25 ◆◆◆

实现一个基于进程的 Tiny Web 服务器的并发版本。你的解答应该为每一个新的连接请求创建一

个新的子进程。使用一个实际的 Web 浏览器来测试你的解答。

code/conc/tfgets-main.c

```
1   #include "csapp.h"
2
3   char *tfgets(char *s, int size, FILE *stream);
4
5   int main()
6   {
7       char buf[MAXLINE];
8
9       if (tfgets(buf, MAXLINE, stdin) == NULL)
10          printf("BOOM!\n");
11      else
12          printf("%s", buf);
13
14      exit(0);
15  }
```

code/conc/tfgets-main.c

图 13.42 习题 13.22～13.24 的驱动程序

13.26 ◆◆◆

实现一个基于 I/O 多路复用的 Tiny Web 服务器的并发版本。使用一个实际的浏览器来测试你的解答。

13.27 ◆◆◆

实现一个基于线程的 Tiny Web 服务器的并发版本。你的解答应该为每一个新的连接请求创建一个新的线程。使用一个实际的浏览器来测试你的解答。

13.28 ◆◆◆◆

实现一个 Tiny Web 服务器的并发预线程化的版本。你的解答应该根据当前的负载，动态地增加或减少线程的数目。一个策略是当缓冲区变满时，将线程数量翻倍，而当缓冲区变为空时，将线程数目减半。使用一个实际的浏览器来测试你的解答。

13.29 ◆◆◆◆

Web 代理是一个在 Web 服务器和浏览器之间扮演中间角色的程序。浏览器不是直接连接服务器以获取网页，而是与代理连接，代理再将请求转发给服务器。当服务器响应代理时，代理将响应发送给浏览器。实现这个试验，请你编写一个简单的可以过滤和记录请求的 Web 代理：

A. 试验的第一部分中，你要建立以接收请求的代理，分析 HTTP，转发请求给服务器，并且返回结果给浏览器。你的代理将所有请求的 URL 记录在磁盘上一个日志文件中，同时它还要阻塞所有

对包含在磁盘上一个过滤文件中的URL的请求。

B. 试验的第二部分中，你要升级你的代理，它通过派生一个独立的线程来处理每一个请求，使得你的代理能够一次处理多个打开的连接。当你的代理在等待远程服务器响应一个请求使它能服务于一个浏览器时，它应该可以处理来自另一个浏览器未完成的请求。

使用一个实际的浏览器来检验你的解答。

练习题答案

练习题 13.1 答案

当父进程派生子进程时，它得到一个已连接描述符的副本，并将相关文件表中的引用计数从1增加到2。当父进程关闭它的描述符副本时，引用计数就从2减少到1。因为内核不会关闭一个文件，直到文件表中它的引用计数值变为零，所以子进程这边的连接端将保持打开。

练习题 13.2 答案

当一个进程因为某种原因终止时，内核将关闭所有打开的描述符。因此，当子进程退出时，它的连接文件描述符的副本也将被自动关闭。

练习题 13.3 答案

回想一下，如果一个从描述符中读一个字节的请求不会阻塞，那么这个描述符就准备好可以读了。假如EOF在一个描述符上为真，那么描述符也准备好可读了，因为读操作将立即返回一个零返回码，表示EOF。因此，键入ctrl-d会导致select函数返回，准备好的集合中有描述符0。

练习题 13.4 答案

因为变量pool.read_set既作为输入参数也作为输出参数，所以我们在每一次调用select之前都重新初始化它。在输入时，它包含读集合。在输出，它包含准备好的集合。

练习题 13.5 答案

因为线程运行在同一个进程中，它们都共享相同的描述符表。无论有多少线程使用这个已连接描述符，这个已连接描述符的文件表的引用计数都等于一。因此，当我们用完它时，一个close操作就足以释放与这个已连接描述符相关的存储器资源了。

练习题 13.6 答案

这里的主要的意思是说，当共享全局和静态变量时，静态变量是私有的。诸如cnt这样的静态变量有点小麻烦，因为共享是限制在它们的函数范围内的——在这个例子中，就是线程例程。

A. 下面就是这张表：

变量名	被主线程引用?	被对等线程0引用?	被对等线程1引用?
ptr	是	是	是
cnt	否	是	是
i.m	是	否	否

（续表）

变量名	被主线程引用？	被对等线程 0 引用？	被对等线程 1 引用？
msgs.m	是	是	是
myid.p0	否	是	否
myid.p1	否	否	是

说明：

- **ptr**：一个被主线程写和被对等线程读的全局变量。
- **cnt**：一个静态变量，被两个对等线程读和写，在存储器中只有一个实例。
- **i.m**：一个存储在主线程栈中的本地自动变量。虽然它的值被传递给对等线程，但是对等线程也决不会在栈中引用它，因此它不是共享的。
- **msgs.m**：一个存储在主线程栈中的本地自动变量，被两个对等线程通过 ptr 间接地引用。
- **myid.0** 和 **myid.1**：分别驻留在对等线程 0 和线程 1 的栈中的一个本地自动变量的实例。

B. 变量 ptr、cnt 和 msgs 被多于一个线程引用，因此它们是共享的。

练习题 13.7 答案

这里的重要思想是，你不能假设当内核调度你的线程时，会如何选择顺序。

步骤	线程	指令	$\%eax_1$	$\%eax_2$	ctr
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	2	H_2	—	—	0
4	2	L_2	—	0	0
5	2	U_2	—	1	0
6	2	S_2	—	1	1
7	1	U_1	1	—	1
8	1	S_1	1	—	1
9	1	T_1	1	—	1
10	2	T_2	1	—	1

变量 cnt 最终有一个不正确的值——1。

练习题 13.8 答案

gethostbyname_ts 函数不是可重入函数，因为每次调用都共享相同的由 gethostbyname 函数返回的 static 变量。然而，它是线程安全的，因为对共享变量的访问是被 *P* 和 *V* 操作保护的，因此是互

序的。

练习题 13.9 答案

如果在第 15 行调用了 pthread_create 之后，我们立即释放块，那么我们将引入一个新的竞争，这次竞争发生在主线程对 free 的调用和线程例程中第 25 行的赋值语句之间。

练习题 13.10 答案

A. 另一种方法是直接传递整数 i，而不是传递一个指向 i 的指针：

```
for (i = 0; i<N;i++)
    Pthread_create(&tid[i],NULL,thread, (void *)i);
```

在线程例程中，我们将参数强制转换成一个 int 类型，并将它赋值给 myid：

```
int myid = (int) vargp;
```

B. 优点是它通过消除对 malloc 和 free 的调用，降低了开销。一个明显的缺点是，它假设指针至少和 int 一样大。即便这种假设对于所有的现代系统来说都为真，但是它对于那些过去遗留下来的或今后的系统来说可能就不为真了。

练习题 13.11 答案

A. 原始的程序的进度图如图 13.43 所示。

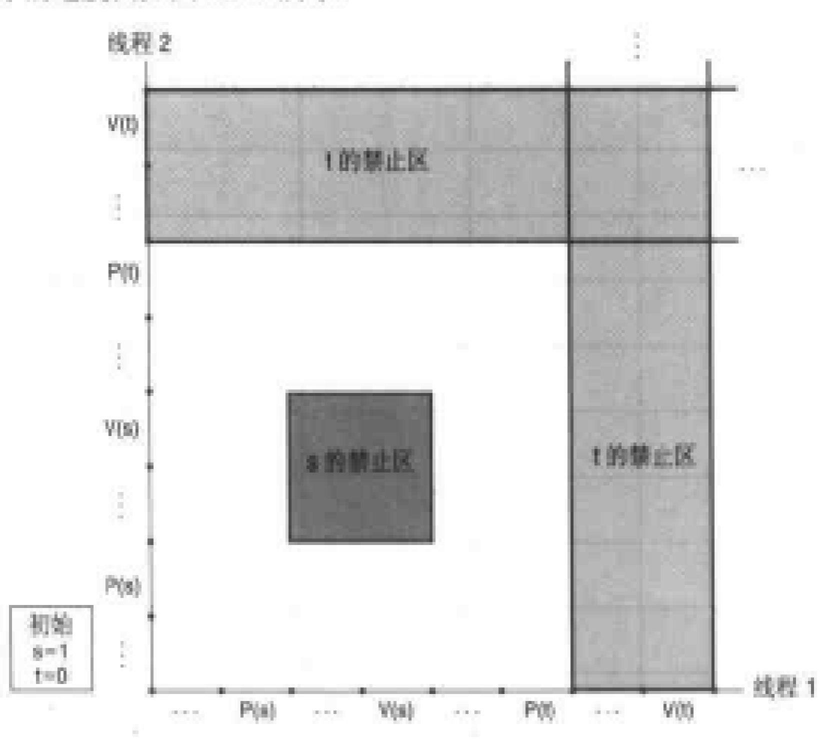


图 13.43 一个会死锁的程序的进度图

B. 因为任何可行的轨迹最终都陷入的死锁状态中，所以这个程序总是会死锁。

C. 为了消除潜在的死锁，将二进制信号量 t 初始化为 1 而不是 0。

D. 正确的程序的进度图如图 13.44 所示。

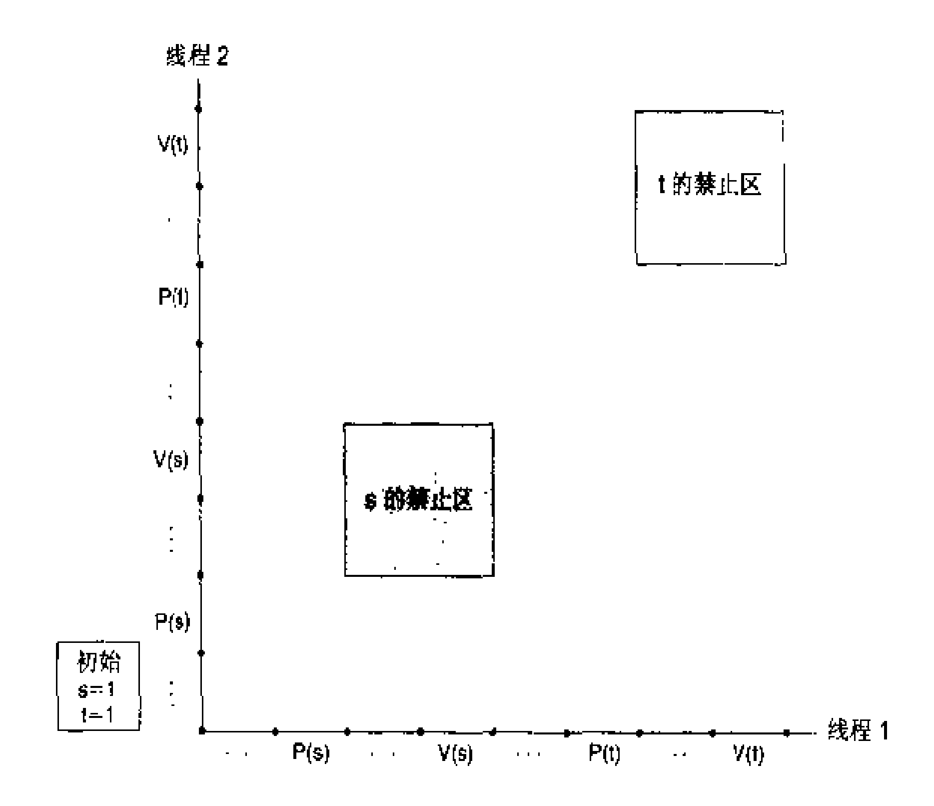


图 13.44 正确的无死锁的程序的进度图

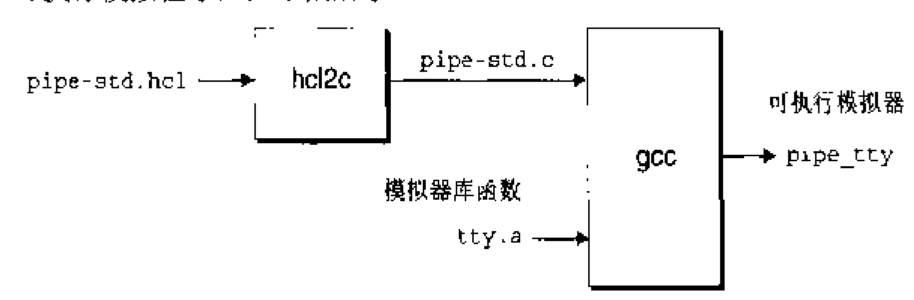
APPENDIX A

处理器控制逻辑的 HCL 描述

A.1 HCL参考手册

在第4章中，我们用HCL（Hardware Control Language，硬件控制语言）来描述了几种处理器设计的控制逻辑部分。HCL具有一些硬件描述语言的属性，允许用户描述布尔函数和字级选择操作。另一方面，它缺乏许多在真正的HDL中能找到的特性，例如，声明寄存器和其他存储元素的方法，循环和条件构造，模块定义和实例化的能力，以及位提取和插入操作。

HCL实际上只是一种语言，用于生成固定格式的C代码。HCL文件中的所有块定义都由程序HCL2C（表示“HCL to C”）转换成C函数。然后再编译这些函数，与实现其他模拟器函数的库代码链接，产生可执行模拟程序，如下图所示：



这张图展示的文件被用来生成流水线模拟器的文本版本。

可以直接用C来描述控制逻辑的行为，而不必写HCL，然后再翻译成C。使用HCL的优点是我们可以更清晰地区分硬件的功能和模拟器的内部工作方式。

HCL只支持两种数据类型：bool（表示“布尔”）信号要么是0，要么是1，而int（表示“整数”）信号等价于C中的int值。数据类型int用于表示所有的多位信号类型，例如，字、寄存器ID和指令代码。当转换成C时，这两种数据类型都表示为int数据，只不过bool类型的值只能等于0或者1。

A.1.1 信号声明

HCL中的表达式可以引用整数或者布尔类型的命名信号。信号名必须以字母（a～z或者A～Z）开头，后面可以是任意数量的字母、数字或者下划线（_）。信号名是大小写敏感的。HCL布尔和整数表达式中的布尔和整数信号名实际上就是C表达式的别名。信号的声明也定义了相关的C表达式。信号声明可以具有如下形式中的一种：

```
boolsig  name  'C-expr'
intsig   name  'C-expr'
```

这里，C-expr可以是任意的C表达式，除了它不能包含单引号（'）或者换行符（\n）以外。当产生C代码时，HCL2C会用相应的C表达式替换所有的信号名。

A.1.2 引号引起来的文本

引号引起来的文本提供了一种从HCL2C直接传递文本到生成的C文件的机制。可以用它来插入变量声明、include语句，以及其他一些通常能在C文件中发现的东西。通用格式为：

```
quote 'string'
```

这里string可以是任何不包含单引号（'）或者换行符（\n）的字符串。

A.1.3 表达式和块

有两种类型的表达式：布尔表达式和整数表达式，在我们的语法描述中分别称为 bool-expr 和 int-expr。图 A.1 列出了不同的布尔表达式类型。按照优先级的降序排列，同一组（组与组之间由水平线分隔）内的操作具有相等的优先级。可以用括号来改变普通的操作符优先级。

最高级是常数值 0 和 1，以及命名的布尔信号。优先级低一级的是以整数为参数但是得到布尔结果的表达式。集合成员关系测试将第一个整数表达式 int-expr 的值与组成集合的每个整数表达式的值{$\text{int-expr}_1,\cdots,\text{int-expr}_k$}相比较，如果发现相匹配的值，结果为 1。关系操作符比较两个整数表达式，当关系满足时，产生 1，当关系不满足时，产生 0。

语 法	含 义
0	逻辑值 0
1	逻辑值 1
name	命名的布尔信号
int-expr in {$\textit{int-expr}_1, \textit{int-expr}_2, \cdots, \textit{int-expr}_k$}	集合成员关系测试
$\textit{int-expr}_1$ == $\textit{int-expr}_2$	相等测试
$\textit{int-expr}_1$!= $\textit{int-expr}_2$	不等测试
$\textit{int-expr}_1$ < $\textit{int-expr}_2$	小于测试
$\textit{int-expr}_1$ <= $\textit{int-expr}_2$	小于或等于测试
$\textit{int-expr}_1$ > $\textit{int-expr}_2$	大于测试
$\textit{int-expr}_1$ >= $\textit{int-expr}_2$	大于或等于测试
!*bool-expr*	Not
$\textit{bool-expr}_1$ && $\textit{bool-expr}_2$	And
$\textit{bool-expr}_1$ || $\textit{bool-expr}_2$	Or

图 A.1 HCL 布尔表达式

这些表达式求值为 0 或者 1。操作是按照优先级的降序排列的，每一组内的操作具有相等的优先级。

图 A.1 中剩下的表达式是由使用布尔连接符的公式组成的（!表示 Not，&&表示 And，而||表示 Or）。

只有三种类型的整数表达式：数字、命名的整数信号和情况（case）表达式。数字是以十进制表示法书写的，可以为负。命名的整数信号使用同前面讲过的一样的命名规则。情况表达式有下面的一般形式：

```
[
        bool-expr1 : int-expr1
        bool-expr2 : int-expr2
        ⋮
        bool-exprk : int-exprk
]
```

表达式包含一系列情况，每种情况 i 是由一个布尔表达式 bool-expr_i 和一个整数表达式 int-expr_i 组成，前者表明是否该选择这种情况，而后者是对于这种情况得到的值。在对一个情况表达式求值

时，布尔表达式是按照顺序被求值的。一旦有一个布尔表达式得到 1，那么相应的整数表达式的值就作为情况表达式的值被返回。如果没有布尔表达式求值为 1，那么这个情况表达式的值就为 0。一个好的编程习惯是让最后一个布尔表达式为 1，以保证至少有一个匹配的情况。

HCL 表达式被用来定义组合逻辑块的行为。块的定义有以下形式之一：

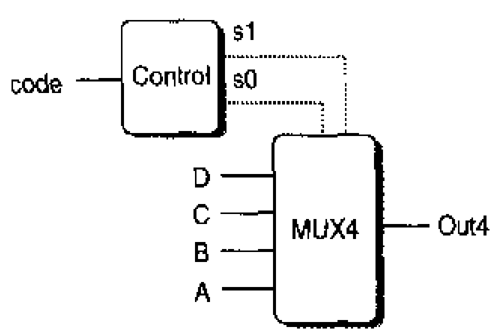
bool *name* = *bool-expr*;

int *name* = *int-expr*;

这里，第一种形式定义的是布尔块，而第二种定义的是字级块。对于一个声明为以 name 为名字的块，HCL2C 产生一个函数 gen_name。这个函数没有参数，而它返回一个 int 类型的结果。

A.1.4 HCL 示例

下面这个示例给出了一个完整的 HCL 文件，用 HCL2C 处理它得到的 C 代码是完全自包含的。可以编译这个代码，并带上表示输入信号的命令行参数运行它。更加典型的情况是，HCL 文件只定义模拟模型的控制部分。然后生成出来的 C 代码被编译，并与其他代码链接，形成可执行模拟器。我们展示这个示例只是为了给出 HCL 的一个具体的例子。该电路是基于 4.2.4 节中描述的 MUX4 电路的，其结构如下：




```
1   ## Simple example of an HCL file.
2   ## This file can be converted to C using hcl2c, and then compiled.
3
4   ## In this example, we will generate the MUX4 circuit shown in
5   ## Section 4.2.4. It consists of a control block that generates
6   ## bit-level signals s1 and s0 from the input signal code,
7   ## and then uses these signals to control a 4-way multiplexor
8   ## with data inputs A, B, C, and D.
9
10  ## This code is embedded in a C program that reads
11  ## the values of code, A, B, C, and D from the command line
12  ## and then prints the circuit output
13
14  ## Information that is inserted verbatim into the C file
15  quote '#include <stdio.h>'
16  quote '#include <stdlib.h>'
17  quote 'int code_val, s0_val, s1_val;'
18  quote 'char **data_names;'
19
```

```
20  ## Declarations of signals used in the HCL description and
21  ## the corresponding C expressions.
22  boolsig s0 's0_val'
23  boolsig s1 's1_val'
24  intsig code 'code_val'
25  intsig A 'atoi(data_names[0])'
26  intsig B 'atoi(data_names[1])'
27  intsig C 'atoi(data_names[2])'
28  intsig D 'atoi(data_names[3])'
29
30  ## HCL descriptions of the logic blocks
31  bool s1 = code in { 2, 3 };
32
33  bool s0 = code in { 1, 3 };
34
35   int Out4 = [
36           !s1 && !s0  : A; # 00
37           !s1         : B; # 01
38           s1 && !s0   : C; # 10
39           1           : D; # 11
40   ];
41
42   ## More information inserted verbatim into the C code to
43   ## compute the values and print the output
44   quote 'int main(int argc, char *argv[]) {'
45   quote 'data_names = argv+2;'
46   quote 'code_val = atoi(argv[1]);'
47   quote 's1_val = gen_s1();'
48   quote 's0_val = gen_s0();'
49   quote 'printf("Out = %d\n", gen_Out4());'
50   quote 'return 0;'
51   quote '}'
```

这个文件定义了布尔信号s0和s1，以及整数信号code，作为对全局变量s0_val、s1_val和code_val引用的别名。它还声明了整数信号 A、B、C 和 D，这里，相应的 C 表达式对于作为命令行参数传递进来的字符串应用标准库函数 atoi。

名字为 s1 的块的定义生成下列 C 代码：

```
int gen_s1()
{
    return ((code_val) == 2||(code_val) == 3);
}
```

从这里可以看出，集合成员关系测试是以一系列的比较来实现的，每次对信号 code 的引用都被替换成了 C 表达式 code_val。

注意，这个HCL文件第23行上声明的信号s1与第31行上声明的名为s1的块之间没有直接的关系。一个是C表达式的别名，而另一个产生名为gen_s1的函数。

最后被引号引起来的文本产生下列主函数：

```
int main(int argc, char *argv[]) {
  data_names = argv+2;
  code_val = atoi(argv[1]);
  s1_val = gen_s1();
  s0_val = gen_s0();
  printf("Out = %d\n", gen_Out4());
  return 0;
}
```

主函数调用函数gen_s1、gen_s0和gen_Out4，这些函数都是根据块定义生成的。我们还可以看出C代码必须如何定义块求值和设置值的顺序，这些被设置的值被用在表示不同信号值的C表达式中。

A.2 SEQ

code/arch/seq/seq-std.hcl

```
1   ####################################################################
2   # HCL Description of Control for Single Cycle Y86 Processor SEQ    #
3   # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002        #
4   ####################################################################
5
6   ####################################################################
7   #    C Include's. Don't alter these                                #
8   ####################################################################
9
10  quote '#include <stdio.h>'
11  quote '#include "isa.h"'
12  quote '#include "sim.h"'
13  quote 'int sim_main(int argc, char *argv[]);'
14  quote 'int gen_pc(){return 0;}'
15  quote 'int main(int argc, char *argv[])'
16  quote '  {plusmode=0;return sim_main(argc,argv);}'
17
18  ####################################################################
19  #    Declarations. Do not change/remove/delete any of these        #
20  ####################################################################
21
22  ##### Symbolic representation of Y86 Instruction Codes #############
23  intsig INOP     'I_NOP'
24  intsig IHALT    'I_HALT'
```

```
25   intsig IRRMOVL    'I_RRMOVL'
26   intsig IIRMOVL    'I_IRMOVL'
27   intsig IRMMOVL    'I_RMMOVL'
28   intsig IMRMOVL    'I_MRMOVL'
29   intsig IOPL       'I_ALU'
30   intsig IJXX       'I_JMP'
31   intsig ICALL  'I_CALL'
32   intsig IRET       'I_RET'
33   intsig IPUSHL 'I_PUSHL'
34   intsig IPOPL  'I_POPL'
35
36    ##### Symbolic representation of Y86 Registers referenced explicitly #####
37    intsig  RESP     'REG_ESP'        # Stack Pointer
38    intsig  RNONE    'REG_NONE'       # Special value indicating "no register"
39
40    ##### ALU Functions referenced explicitly                          #####
41    intsig ALUADD 'A_ADD'             # ALU should add its arguments
42
43    ##### Signals that can be referenced by control logic ####################
44
45    ##### Fetch stage inputs            #####
46   intsig pc 'pc'                     # Program counter
47   ##### Fetch stage computations      #####
48   intsig icode 'icode'               # Instruction control code
49   intsig ifun  'ifun'                # Instruction function
50   intsig rA    'ra'                  # rA field from instruction
51   intsig rB    'rb'                  # rB field from instruction
52   intsig valC  'valc'                # Constant from instruction
53   intsig valP  'valp'                # Address of following instruction
54
55   ##### Decode Stage computations     #####
56   intsig valA    'vala'              # Value from register A port
57   intsig valB    'valb'              # Value from register B port
58
59   ##### Execute stage computations    #####
60   intsig valE    'vale'              # Value computed by ALU
61   boolsig Bch    'bcond'             # Branch test
62
63   ##### Memory stage computations     #####
64   intsig valM  'valm'                # Value read from memory
65
66
67   ####################################################################
68   #       Control Signal Definitions.                                #
69   ####################################################################
```

```
70
71  ################ Fetch Stage ###################################################
72
73   # Does fetched instruction require a regid byte?
74   bool need_regids =
75            icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
76                           IIRMOVL, IRMMOVL, IMRMOVL };
77
78   # Does fetched instruction require a constant word?
79   bool need_valC =
80            icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
81
82   bool instr_valid = icode in
83            { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
84                   IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
85
86   ################ Decode Stage ###################################
87
88   ## What register should be used as the A source?
89   int srcA = [
90           icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
91           icode in { IPOPL, IRET } : RESP;
92           1 : RNONE; # Don't need register
93   ];
94
95   ## What register should be used as the B source?
96   int srcB = [
97          icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
98          icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
99          1 : RNONE; # Don't need register
100  ];
101
102  ## What register should be used as the E destination?
103  int dstE = [
104          icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
105          icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
106          1 : RNONE; # Don't need register
107  ];
108
109  ## What register should be used as the M destination?
110  int dstM = [
111           icode in { IMRMOVL, IPOPL } : rA;
112           1 : RNONE; # Don't need register
113  ];
114
```

```
115  ############### Execute Stage ####################################
116
117  ## Select input A to ALU
118  int aluA = [
119          icode in { IRRMOVL, IOPL } : valA;
120          icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
121          icode in { ICALL, IPUSHL } : -4;
122          icode in { IRET, IPOPL } : 4;
123          # Other instructions don't need ALU
124  ];
125
126  ## Select input B to ALU
127  int aluB = [
128          icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
129                        IPUSHL, IRET, IPOPL } : valB;
130          icode in { IRRMOVL, IIRMOVL } : 0;
131          # Other instructions don't need ALU
132  ];
133
134  ## Set the ALU function
135  int alufun = [
136          icode == IOPL : ifun;
137          1 : ALUADD;
138  ];
139
140  ## Should the condition codes be updated?
141 bool set_cc = icode in { IOPL };
142
143  ############### Memory Stage #####################################
144
145  ## Set read control signal
146  bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
147
148  ## Set write control signal
149  bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
150
151  ## Select memory address
152  int mem_addr = [
153          icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
154          icode in { IPOPL, IRET } : valA;
155          # Other instructions don't need address
156  ];
157
158  ## Select memory input data
159  int mem_data = [
```

```
160         # Value from register
161         icode in { IRMMOVL, IPUSHL } : valA;
162         # Return PC
163         icode == ICALL : valP;
164         # Default: Don't write anything
165   ];
166
167   ################ Program Counter Update ############################
168
169   ## What address should instruction be fetched at
170
171   int new_pc = [
172         # Call. Use instruction constant
173         icode == ICALL : valC;
174         # Taken branch.  Use instruction constant
175         icode == IJXX && Bch : valC;
176         # Completion of RET instruction. Use value from stack
177         icode == IRET : valM;
178         # Default: Use incremented PC
179         1 : valP;
180   ];
```

code/arch/seq/seq-std.hcl

A.3 SEQ+

code/arch/seq/seq+-std.hcl

```
1     ############### ################################################
2     # HCL Description of Control for Single Cycle Y86 Processor SEQ+ #
3     # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002      #
4     ##################################################################
5
6     ##################################################################
7     #    C Include's. Don't alter these                              #
8     ##################################################################
9
10    quote '#include <stdio.h>'
11    quote '#include "isa.h"'
12    quote '#include "sim.h"'
13    quote 'int sim_main(int argc, char *argv[]);'
14    quote 'int gen_new_pc(){return 0;} '
15    quote 'int main(int argc, char *argv[])'
16    quote '  {plusmode=1;return sim_main(argc,argv);}'
17
18    ##################################################################
```

```
19   #     Declarations. Do not change/remove/delete any of these              #
20   ##############################################################################
21
22   ##### Symbolic representation of Y86 Instruction Codes ############
23   intsig INOP     'I_NOP'
24   intsig IHALT    'I_HALT'
25   intsig IRRMOVL  'I_RRMOVL'
26   intsig IIRMOVL  'I_IRMOVL'
27   intsig IRMMOVL  'I_RMMOVL'
28   intsig IMRMOVL  'I_MRMOVL'
29   intsig IOPL     'I_ALU'
30   intsig IJXX     'I_JMP'
31   intsig ICALL    'I_CALL'
32   intsig IRET     'I_RET'
33   intsig IPUSHL   'I_PUSHL'
34   intsig IPOPL    'I_POPL'
35
36   ##### Symbolic representation of Y86 Registers referenced explicitly ####
37   intsig RESP     'REG_ESP'   # Stack Pointer
38   intsig RNONE    'REG_NONE'  # Special value indicating "no register"
39
40   ##### ALU Functions referenced explicitly                          #####
41   intsig ALUADD   'A_ADD'     # ALU should add its arguments
42
43   ##### Signals that can be referenced by control logic ##################
44
45   ##### PC stage inputs              #####
46
47   ## All of these values are based on those from previous instruction
48   intsig pIcode 'prev_icode'          # Instr. control code
49   intsig pValC 'prev_valc'            # Constant from instruction
50   intsig pValM 'prev_valm'            # Value read from memory
51   intsig pValP 'prev_valp'            # Incremented program counter
52   boolsig pBch 'prev_bcond'           # Branch taken flag
53
54   ##### Fetch stage computations      #####
55   intsig icode 'icode'                # Instruction control code
56   intsig ifun 'ifun'                  # Instruction function
57   intsig rA 'ra'                      # rA field from instruction
58   intsig rB 'rb'                      # rB field from instruction
59   intsig valC 'valc'                  # Constant from instruction
60   intsig valP 'valp'                  # Address of following instruction
61
62   ##### Decode stage computations     #####
63   intsig valA 'vala'                  # Value from register A port
```

```
 64   intsig valB 'valb'                 # Value from register B port
 65
 66   ##### Execute stage computations   #####
 67   intsig valE 'vale'                 # Value computed by ALU
 68   boolsig Bch 'bcond'                # Branch test
 69
 70   ##### Memory stage computations    #####
 71   intsig valM 'valm'                 # Value read from memory
 72
 73
 74   ##############################################################################
 75   #     Control Signal Definitions.                                            #
 76   ##############################################################################
 77
 78   ################ Program Counter Computation      #######################
 79
 80   # Compute fetch location for this instruction based on results from
 81   # previous instruction.
 82
 83   int pc = [
 84         # Call. Use instruction constant
 85         pIcode == ICALL : pValC;
 86         # Taken branch. Use instruction constant
 87         pIcode == IJXX && pBch : pValC;
 88         # Completion of RET instruction. Use value from stack
 89         pIcode == IRET : pValM;
 90         # Default: Use incremented PC
 91         1 : pValP;
 92   ];
 93
 94   ################ Fetch Stage      ###################################
 95
 96   # Does fetched instruction require a regid byte?
 97   bool need_regids =
 98          icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
 99                       IIRMOVL, IRMMOVL, IMRMOVL };
100
101   # Does fetched instruction require a constant word?
102   bool need_valC =
103          icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
104
105   bool instr_valid = icode in
106          { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
107                  IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
108
```

```
109  ################ Decode Stage ###################################
110
111  ## What register should be used as the A source?
112  int srcA = [
113          icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
114          icode in { IPOPL, IRET } : RESP;
115          1 : RNONE; # Don't need register
116  ];
117
118  ## What register should be used as the B source?
119  int srcB = [
120          icode in { IOPL, IRMMOVL, IMRMOVL } : rB;
121          icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
122          1 : RNONE; # Don't need register
123  ];
124
125  ## What register should be used as the E destination?
126  int dstE = [
127          icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
128          icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
129          1 : RNONE; # Don't need register
130  ];
131
132  ## What register should be used as the M destination?
133  int dstM = [
134          icode in { IMRMOVL, IPOPL } : rA;
135          1 : RNONE; # Don't need register
136  ];
137
138  ################ Execute Stage ###################################
139
140  ## Select input A to ALU
141  int aluA = [
142          icode in { IRRMOVL, IOPL } : valA;
143          icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
144          icode in { ICALL, IPUSHL } : -4;
145          icode in { IRET, IPOPL } : 4;
146          # Other instructions don't need ALU
147  ];
148
149  ## Select input B to ALU
150  int aluB = [
151          icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
152                        IPUSHL, IRET, IPOPL } : valB;
153          icode in { IRRMOVL, IIRMOVL } : 0;
```

```
154         # Other instructions don't need ALU
155 ];
156
157 ## Set the ALU function
158 int alufun = [
159         icode == IOPL : ifun;
160         1 : ALUADD;
161 ];
162
163 ## Should the condition codes be updated?
164 bool set_cc = icode in { IOPL };
165
166 ################ Memory Stage ####################################
167
168 ## Set read control signal
169 bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
170
171 ## Set write control signal
172 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
173
174 ## Select memory address
175 int mem_addr = [
176         icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
177         icode in { IPOPL, IRET } : valA;
178         # Other instructions don't need address
179 ];
180
181 ## Select memory input data
182 int mem_data = [
183         # Value from register
184         icode in { IRMMOVL, IPUSHL } : valA;
185         # Return PC
186         icode == ICALL : valP;
187         # Default: Don't write anything
188 ];
```

code/arch/seq/seq+-std.hcl

A.4 PIPE

code/arch/pipe/pipe-std.hcl

```
1   #################################################################
2   # HCL Description of Control for Pipelined Y86 Processor        #
3   # Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002     #
4   #################################################################
```

```
5
6     ####################################################################
7     # C Include's. Don't alter these                                   #
8     ####################################################################
9
10    quote '#include <stdio.h>'
11    quote '#include "isa.h"'
12    quote '#include "pipeline.h"'
13    quote '#include "stages.h"'
14    quote '#include "sim.h"'
15    quote 'int sim_main(int argc, char *argv[]);'
16    quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
17
18    ####################################################################
19    #    Declarations. Do not change/remove/delete any of these        #
20    ####################################################################
21
22    ##### Symbolic representation of Y86 Instruction Codes     ##########
23    intsig INOP         'I_NOP'
24    intsig IHALT        'I_HALT'
25    intsig IRRMOVL      'I_RRMOVL'
26    intsig IIRMOVL      'I_IRMOVL'
27    intsig IRMMOVL      'I_RMMOVL'
28    intsig IMRMOVL      'I_MRMOVL'
29    intsig IOPL         'I_ALU'
30    intsig IJXX         'I_JMP'
31    intsig ICALL        'I_CALL'
32    intsig IRET         'I_RET'
33    intsig IPUSHL       'I_PUSHL'
34    intsig IPOPL        'I_POPL'
35
36    ##### Symbolic representation of Y86 Registers referenced explicitly #####
37    intsig RESP         'REG_ESP'   # Stack Pointer
38    intsig RNONE        'REG_NONE'  # Special value indicating "no register"
39
40    ##### ALU Functions referenced explicitly    ######################
41    intsig ALUADD       'A_ADD'     # ALU should add its arguments
42
43    ##### Signals that can be referenced by control logic    ###########
44
45    ##### Pipeline Register F    ######################################
46
47    intsig F_predPC     'pc_curr->pc'   # Predicted value of PC
48
49    ##### Intermediate Values in Fetch Stage   #########################
```

```
50
51    intsig f_icode 'if_id_next->icode' # Fetched instruction code
52    intsig f_ifun  'if_id_next->ifun'  # Fetched instruction function
53    intsig f_valC  'if_id_next->valc'  # Constant data of fetched instruction
54    intsig f_valP  'if_id_next->valp'  # Address of following instruction
55
56    ##### Pipeline Register D ##########################################
57    intsig D_icode 'if_id_curr->icode'    # Instruction code
58    intsig D_rA    'if_id_curr->ra'       # rA field from instruction
59    intsig D_rB    'if_id_curr->rb'       # rB field from instruction
60    intsig D_valP  'if_id_curr->valp'     # Incremented PC
61
62    ##### Intermediate Values in Decode Stage #########################
63
64    intsig d_srcA  'id_ex_next->srca'     # srcA from decoded instruction
65    intsig d_srcB  'id_ex_next->srcb'     # srcB from decoded instruction
66    intsig d_rvalA 'd_regvala'            # valA read from register file
67     intsig d_rvalB 'd_regvalb'           # valB read from register file
68
69    ##### Pipeline Register E ###########################################
70    intsig E_icode 'id_ex_curr->icode'    # Instruction code
71    intsig E_ifun  'id_ex_curr->ifun'     # Instruction function
72    intsig E_valC  'id_ex_curr->valc'     # Constant data
73    intsig E_srcA  'id_ex_curr->srca'     # Source A register ID
74    intsig E_valA  'id_ex_curr->vala'     # Source A value
75    intsig E_srcB  'id_ex_curr->srcb'     # Source B register ID
76    intsig E_valB  'id_ex_curr->valb'     # Source B value
77    intsig E_dstE  'id_ex_curr->deste'    # Destination E register ID
78    intsig E_dstM  'id_ex_curr->destm'    # Destination M register ID
79
80    ##### Intermediate Values in Execute Stage########################
81    intsig e_valE 'ex_mem_next->vale'          # valE generated by ALU
82    boolsig e_Bch 'ex_mem_next->takebranch'    # Am I about to branch?
83
84    ##### Pipeline Register M                    #####
85    intsig M_icode 'ex_mem_curr->icode'        # Instruction code
86    intsig M_ifun 'ex_mem_curr->ifun'          # Instruction function
87    intsig M_valA 'ex_mem_curr->vala'          # Source A value
88    intsig M_dstE 'ex_mem_curr->deste'         # Destination E register ID
89    intsig M_valE 'ex_mem_curr->vale'          # ALU E value
90    intsig M_dstM 'ex_mem_curr->destm'         # Destination M register ID
91    boolsig M_Bch 'ex_mem_curr->takebranch'    # Branch Taken flag
92
93    ##### Intermediate Values in Memory Stage #########################
94    intsig m_valM 'mem_wb_next->valm'      # valM generated by memory
```

```
95
96   ##### Pipeline Register W ##########################################
97   intsig W_icode 'mem_wb_curr->icode'   # Instruction code
98   intsig W_dstE 'mem_wb_curr->deste'    # Destination E register ID
99   intsig W_valE 'mem_wb_curr->vale'     # ALU E value
100  intsig W_dstM 'mem_wb_curr->destm'    # Destination M register ID
101  intsig W_valM 'mem_wb_curr->valm'     # Memory M value
102
103  ####################################################################
104  #     Control Signal Definitions.                                  #
105  ####################################################################
106
107  ################ Fetch Stage ###################################
108
109  ## What address should instruction be fetched at
110  int f_pc = [
111          # Mispredicted branch. Fetch at incremented PC
112      M_icode == IJXX && !M_Bch : M_valA;
113      # Completion of RET instruction.
114      W_icode == IRET : W_valM;
115      # Default: Use predicted value of PC
116      1 : F_predPC;
117  ];
118
119  # Does fetched instruction require a regid byte?
120  bool need_regids =
121          f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
122                       IIRMOVL, IRMMOVL, IMRMOVL };
123
124  # Does fetched instruction require a constant word?
125  bool need_valC =
126          f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
127
128  bool instr_valid = f_icode in
129          { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
130                  IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
131
132  # Predict next value of PC
133  int new_F_predPC = [
134          f_icode in { IJXX, ICALL } : f_valC;
135          1 : f_valP;
136  ];
137
138
139  ################ Decode Stage ###################################
```

```
140
141
142    ## What register should be used as the A source?
143    int new_E_srcA = [
144            D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : D_rA;
145            D_icode in { IPOPL, IRET } : RESP;
146            1 : RNONE; # Don't need register
147    ];
148
149    ## What register should be used as the B source?
150    int new_E_srcB = [
151            D_icode in { IOPL, IRMMOVL, IMRMOVL } : D_rB;
152            D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
153            1 : RNONE; # Don't need register
154    ];
155
156    ## What register should be used as the E destination?
157    int new_E_dstE = [
158            D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
159            D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
160            1 : RNONE; # Don't need register
161 ];
162
163    ## What register should be used as the M destination?
164    int new_E_dstM = [
165    D_icode in { IMRMOVL, IPOPL } : D_rA;
166     1 : RNONE; # Don't need register
167    ];
168
169    ## What should be the A value?
170    ## Forward into decode stage for valA
171    int new_E_valA = [
172            D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
173            d_srcA == E_dstE : e_valE; # Forward valE from execute
174            d_srcA == M_dstM : m_valM; # Forward valM from memory
175            d_srcA == M_dstE : M_valE; # Forward valE from memory
176            d_srcA == W_dstM : W_valM; # Forward valM from write back
177            d_srcA == W_dstE : W_valE; # Forward valE from write back
178            1 : d_rvalA; # Use value read from register file
179    ];
180
181     int new_E_valB = [
182             d_srcB == E_dstE : e_valE; # Forward valE from execute
183             d_srcB == M_dstM : m_valM; # Forward valM from memory
184             d_srcB == M_dstE : M_valE; # Forward valE from memory
```

```
185                d_srcB == W_dstM : W_valM;  # Forward valM from write back
186                d_srcB == W_dstE : W_valE;  # Forward valE from write back
187                1 : d_rvalB; # Use value read from register file
188     ];
189
190     ################ Execute Stage #####################################
191
192     ## Select input A to ALU
193     int aluA = [
194               E_icode in { IRRMOVL, IOPL } : E_valA;
195               E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
196               E_icode in { ICALL, IPUSHL } : -4;
197               E_icode in { IRET, IPOPL } : 4;
198               # Other instructions don't need ALU
199     ];
200
201     ## Select input B to ALU
202     int aluB = [
203               E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
204                              IPUSHL, IRET, IPOPL } : E_valB;
205               E_icode in { IRRMOVL, IIRMOVL } : 0;
206               # Other instructions don't need ALU
207     ];
208
209     ## Set the ALU function
210     int alufun = [
211                E_icode == IOPL : E_ifun;
212                1 : ALUADD;
213     ];
214
215     ## Should the condition codes be updated?
216     bool set_cc = E_icode == IOPL;
217
218
219     ################ Memory Stage ######################################
220
221     ## Select memory address
222     int mem_addr = [
223     M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
224                M_icode in { IPOPL, IRET } : M_valA;
225            # Other instructions don't need address
```

```
226   ];
227
228   ## Set read control signal
229   bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
230
231   ## Set write control signal
232   bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
233
234
235   ################ Pipeline Register Control #########################
236
237   # Should I stall or inject a bubble into Pipeline Register F?
238   # At most one of these can be true.
239   bool F_bubble = 0;
240   bool F_stall =
241           # Conditions for a load/use hazard
242           E_icode in { IMRMOVL, IPOPL } &&
243           E_dstM in { d_srcA, d_srcB } ||
244           # Stalling at fetch while ret passes through pipeline
245           IRET in { D_icode, E_icode, M_icode };
246
247   # Should I stall or inject a bubble into Pipeline Register D?
248   # At most one of these can be true.
249   bool D_stall =
250           # Conditions for a load/use hazard
251           E_icode in { IMRMOVL, IPOPL } &&
252            E_dstM in { d_srcA, d_srcB };
253
254   bool D_bubble =
255           # Mispredicted branch
256           (E_icode == IJXX && !e_Bch) ||
257           # Stalling at fetch while ret passes through pipeline
258           IRET in { D_icode, E_icode, M_icode };
259
260   # Should I stall or inject a bubble into Pipeline Register E?
261   # At most one of these can be true.
262   bool E_stall = 0;
263   bool E_bubble =
264           # Mispredicted branch
265           (E_icode == IJXX && !e_Bch) ||
266           # Conditions for a load/use hazard
```

```
267             E_icode in { IMRMOVL, IPOPL } &&
268                E_dstM in { d_srcA, d_srcB};
269
270     # Should I stall or inject a bubble into Pipeline Register M?
271     # At most one of these can be true.
272     bool M_stall = 0;
273     bool M_bubble = 0;
```

code/arch/pipe/pipe-std.hcl

APPENDIX **B**

错误处理

程序员应该总是检查系统级函数返回的错误代码。有许多细微方式导致错误的出现，只有使用内核能够提供给我们的状态信息才能理解为什么有这样的错误。不幸的是，程序员往往不愿意进行错误检查，因为这使他们的代码变得很庞大，将一行代码变成一个多行的条件语句。错误检查也是很令人迷惑的，因为不同的函数表示不同方面的错误。

在编写本书时，我们面临类似的问题。一方面，我们希望我们的代码示例阅读起来简洁简单。另一方面，我们又不希望给学生们一个错误的印象，以为可以省略错误检查。为了解决这些问题，我们采用了一种基于错误处理包装函数（error-handling wrapper）的方法，这是由 W. Richard Stevens 在他的网络编程教材[81]中最先提出的。

其思想是，给定某个基本的系统级函数 foo，我们定义一个有相同参数、只不过开头字母大写了的包装函数 Foo。包装函数调用基本函数，并检查错误。如果包装函数发现了错误，那么它就打印一条信息，并终止进程。否则，它返回到调用者。注意，如果没有错误，包装函数的行为与基本函数完全一样。换句话说，如果程序使用包装函数运行正确，那么我们把每个包装函数的第一个字母小写并重新编译，也能正确运行。

包装函数被封装在一个源文件（csapp.c）中，这个文件被编译和链接到每个程序中。一个独立的头文件（csapp.h）中包含这些包装函数的函数原型。

本附录给出了一个关于 Unix 系统中不同种类的错误处理的指南，还给出了不同风格的错误处理包装函数的示例。为了方便参考，我们还包括了 csapp.h 和 csapp.c 文件的完整源代码。

B.1 Unix 系统中的错误处理

本书中我们遇到的系统级函数调用使用三种不同风格的返回错误：Unix 风格的、Posix 风格的和 DNS 风格的。

Unix 风格的错误处理

像 fork 和 wait 这样 Unix 早期开发出来的函数（以及一些较老的 Posix 函数）的函数返回值既包括错误代码，也包括有用的结果。例如，当 Unix 风格的 wait 函数遇到一个错误（例如没有了进程要回收），它就返回−1，并将全局变量 errno 设置为指明错误原因的错误代码。如果 wait 成功完成，那么它就返回有用的结果，也就是回收的子进程的 PID。Unix 风格的错误处理代码通常具有以下形式：

```
1    if ((pid = wait(NULL)) < 0) {
2        fprintf(stderr, "wait error: %s\n", strerror(errno));
3        exit(0);
4    }
```

strerror 函数返回某个 errno 值的文本描述。

Posix 风格的错误处理

许多较新的 Posix 函数，例如 Pthread 函数，只用返回值来表明成功（0）或者失败（非 0）。任何有用的结果都返回在通过引用传递进来的函数参数中。我们称这种方法为 Posix 风格的错误处理。例如，Posix 风格的 pthread_create 函数用它的返回值来表明成功或者失败，而通过引用将新创建的

线程的 ID（有用的结果）返回放在它的第一个参数中。Posix 风格的错误处理代码通常具有以下形式：

```
1   if ((retcode = pthread_create(&tid, NULL, thread, NULL)) != 0) {
2       fprintf(stderr, "pthread_create error: %s\n", strerror(retcode));
3       exit(0);
4   }
```

DNS 风格的错误处理

gethostbyname 和 gethostbyaddr 函数检索 DNS（域名系统）主机条目，它们有另外一种返回错误的方法。这些函数在失败时返回 NULL 指针，并设置全局变量 h_errno。DNS 风格的错误处理通常具有以下形式：

```
1   if ((p = gethostbyname(name)) == NULL) {
2       fprintf(stderr, "gethostbyname error: %s\n:", hstrerror(h_errno));
3       exit(0);
4   }
```

错误报告函数小结

贯穿本书，我们使用下列错误报告函数来包容不同的错误处理风格：

```
#include "csapp.h"

void unix_error(char *msg);
void posix_error(int code, char *msg);
void dns_error(char *msg);
void app_error(char *msg);
```

返回：无。

正如它们的名字表明的那样，unix_error、posix_error 和 dns_error 函数报告 Unix 风格的错误、Posix 风格的错误和 DNS 风格的错误，然后终止。app_error 函数是为了方便报告应用错误。它只是简单地打印它的输入，然后终止。图 B.1 展示了这些错误报告函数的代码。

code/src/csapp.c

```
1   void unix_error(char *msg) /* unix-style error */
2   {
3       fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4       exit(0);
5   }
6
7   void posix_error(int code, char *msg) /* posix-style error */
8   {
9       fprintf(stderr, "%s: %s\n", msg, strerror(code));
10      exit(0);
11  }
12
13  void dns_error(char *msg) /* dns-style error */
14  {
```

```
15        fprintf(stderr, "%s: DNS error %d\n", msg, h_errno);
16        exit(0);
17    }
18
19    void app_error(char *msg) /* application error */
20    {
21        fprintf(stderr, "%s\n", msg);
22        exit(0);
23    }
```

code/src/csapp.c

图 B.1 错误报告函数

code/src/csapp.c

```
1    pid_t Wait(int *status)
2    {
3        pid_t pid;
4
5        if ((pid = wait(status)) < 0)
6            unix_error("Wait error");
7        return pid;
8    }
```

code/src/csapp.c

图 B.2 Unix 风格的 wait 函数的包装函数

B.2 错误处理包装函数

下面是一些不同错误处理包装函数的示例：

Unix 风格的错误处理包装函数

图 B.2 展示了 Unix 风格的 wait 函数的包装函数。如果 wait 返回一个错误，包装函数打印一条消息，然后退出。否则，它向调用者返回一个 PID。

图 B.3 展示了 Unix 风格的 kill 函数的包装函数。注意，这个函数同 wait 不同，成功时返回 void。

code/src/csapp.c

```
1    void Kill(pid_t pid, int signum)
2    {
3        int rc;
4
5        if ((rc = kill(pid, signum)) < 0)
6            unix_error("Kill error");
7    }
```

code/src/csapp.c

图 B.3 Unix 风格的 kill 函数的包装函数

Posix 风格的错误处理包装函数

图 B.4 展示了 Posix 风格的 pthread_detach 函数的包装函数。同大多数 Posix 风格的函数一样，它的错误返回码中不会包含有用的结果，所以成功时，包装函数返回 void。

code/src/csapp.c

```
1   void Pthread_detach(pthread_t tid) {
2       int rc;
3
4       if ((rc = pthread_detach(tid)) != 0)
5           posix_error(rc, "Pthread_detach error");
6   }
```

code/src/csapp.c

图 B.4 Posix 风格的 pthread_detach 函数的包装函数

DNS 风格的错误处理包装函数

图 B.5 展示了 DNS 风格的 gethostbyname 函数的包装函数。

code/src/csapp.c

```
1   struct hostent *Gethostbyname(const char *name)
2   {
3       struct hostent *p;
4
5       if ((p = gethostbyname(name)) == NULL)
6           dns_error("Gethostbyname error");
7       return p;
8   }
```

code/src/csapp.c

图 B.5 DNS 风格的 gethostbyname 函数的包装函数

B.3 csapp.h 头文件

code/include/csapp.h

```
1   #ifndef __CSAPP_H__
2   #define __CSAPP_H__
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <unistd.h>
7   #include <string.h>
8   #include <ctype.h>
9   #include <setjmp.h>
10  #include <signal.h>
11  #include <sys/time.h>
12  #include <sys/types.h>
13  #include <sys/wait.h>
```

```
14  #include <sys/stat.h>
15  #include <fcntl.h>
16  #include <sys/mman.h>
17  #include <errno.h>
18  #include <math.h>
19  #include <pthread.h>
20  #include <semaphore.h>
21  #include <sys/socket.h>
22  #include <netdb.h>
23  #include <netinet/in.h>
24  #include <arpa/inet.h>
25
26
27  /* Default file permissions are DEF_MODE & ~DEF_UMASK */
28  #define DEF_MODE  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
29  #define DEF_UMASK S_IWGRP|S_IWOTH
30
31  /* Simplifies calls to bind(), connect(), and accept() */
32  typedef struct sockaddr SA;
33
34  /* Persistent state for the robust I/O (Rio) package */
35  #define RIO_BUFSIZE 8192
36  typedef struct {
37      int rio_fd;                  /* descriptor for this internal buf */
38      int rio_cnt;                 /* unread bytes in internal buf */
39      char *rio_bufptr;            /* next unread byte in internal buf */
40      char rio_buf[RIO_BUFSIZE];   /* internal buffer */
41  } rio_t;
42
43  /* External variables */
44  extern int h_errno;       /* defined by BIND for DNS errors */
45  extern char **environ;    /* defined by libc */
46
47  /* Misc constants */
48  #define MAXLINE 8192      /* max text line length */
49  #define MAXBUF 8192       /* max I/O buffer size */
50  #define LISTENQ 1024      /* second argument to listen() */
51
52  /* Our own error-handling functions */
53  void unix_error(char *msg);
54  void posix_error(int code, char *msg);
55  void dns_error(char *msg);
56  void app_error(char *msg);
57
58  /* Process control wrappers */
```

```
59   pid_t Fork(void);
60   void Execve(const char *filename, char *const argv[], char *const envp[]);
61   pid_t Wait(int *status);
62   pid_t Waitpid(pid_t pid, int *iptr, int options);
63   void Kill(pid_t pid, int signum);
64   unsigned int Sleep(unsigned int secs);
65   void Pause(void);
66   unsigned int Alarm(unsigned int seconds);
67   void Setpgid(pid_t pid, pid_t pgid);
68   pid_t Getpgrp();
69
70   /* Signal wrappers */
71   typedef void handler_t(int);
72   handler_t *Signal(int signum, handler_t *handler);
73   void Sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
74   void Sigemptyset(sigset_t *set);
75   void Sigfillset(sigset_t *set);
76   void Sigaddset(sigset_t *set, int signum);
77   void Sigdelset(sigset_t *set, int signum);
78   int Sigismember(const sigset_t *set, int signum);
79
80   /* Unix I/O wrappers */
81   int Open(const char *pathname, int flags, mode_t mode);
82   ssize_t Read(int fd, void *buf, size_t count);
83   ssize_t Write(int fd, const void *buf, size_t count);
84   off_t Lseek(int fildes, off_t offset, int whence);
85   void Close(int fd);
86   int Select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
87              struct timeval *timeout);
88   int Dup2(int fd1, int fd2);
89   void Stat(const char *filename, struct stat *buf);
90   void Fstat(int fd, struct stat *buf) ;
91
92   /* Memory mapping wrappers */
93   void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
94   void Munmap(void *start, size_t length);
95
96   /* Standard I/O wrappers */
97   void Fclose(FILE *fp);
98   FILE *Fdopen(int fd, const char *type);
99   char *Fgets(char *ptr, int n, FILE *stream);
100  FILE *Fopen(const char *filename, const char *mode);
101  void Fputs(const char *ptr, FILE *stream);
102  size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
103  void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

```
104
105 /* Dynamic storage allocation wrappers */
106 void *Malloc(size_t size);
107 void *Realloc(void *ptr, size_t size);
108 void *Calloc(size_t nmemb, size_t size);
109 void Free(void *ptr);
110
111 /* Sockets interface wrappers */
112 int Socket(int domain, int type, int protocol);
113 void Setsockopt(int s, int level, int optname, const void *optval, int optlen);
114 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen);
115 void Listen(int s, int backlog);
116 int Accept(int s, struct sockaddr *addr, int *addrlen);
117 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
118
119 /* DNS wrappers */
120 struct hostent *Gethostbyname(const char *name);
121 struct hostent *Gethostbyaddr(const char *addr, int len, int type);
122
123 /* Pthreads thread control wrappers */
124 void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
125                     void * (*routine)(void *), void *argp);
126 void Pthread_join(pthread_t tid, void **thread_return);
127 void Pthread_cancel(pthread_t tid);
128 void Pthread_detach(pthread_t tid);
129 void Pthread_exit(void *retval);
130 pthread_t Pthread_self(void);
131 void Pthread_once(pthread_once_t *once_control, void (*init_function)());
132
133 /* POSIX semaphore wrappers */
134 void Sem_init(sem_t *sem, int pshared, unsigned int value);
135 void P(sem_t *sem);
136 void V(sem_t *sem);
137
138 /* Rio (Robust I/O) package */
139 ssize_t rio_readn(int fd, void *usrbuf, size_t n);
140 ssize_t rio_writen(int fd, void *usrbuf, size_t n);
141 void rio_readinitb(rio_t *rp, int fd);
142 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
143 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
144
145 /* Wrappers for Rio package */
146 ssize_t Rio_readn(int fd, void *usrbuf, size_t n);
147 void Rio_writen(int fd, void *usrbuf, size_t n);
148 void Rio_readinitb(rio_t *rp, int fd);
```

```
149 ssize_t Rio_readnb(rio_t *rp, void *usrbuf, size_t n);
150 ssize_t Rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
151
152 /* Client/server helper functions */
153 int open_clientfd(char *hostname, int portno);
154 int open_listenfd(int portno);
155
156 /* Wrappers for client/server helper functions */
157 int Open_clientfd(char *hostname, int port);
158 int Open_listenfd(int port);
159
160 #endif /* __CSAPP_H__ */
```

code/include/csapp.h

B.4 csapp.c 源文件

code/src/csapp.c

```
1   #include "csapp.h"
2
3   /**************************
4    * Error-handling functions
5    **************************/
6   void unix_error(char *msg) /* unix-style error */
7   {
8       fprintf(stderr, "%s: %s\n", msg, strerror(errno));
9       exit(0);
10  }
11
12  void posix_error(int code, char *msg) /* posix-style error */
13  {
14      fprintf(stderr, "%s: %s\n", msg, strerror(code));
15      exit(0);
16  }
17
18  void dns_error(char *msg) /* dns-style error */
19  {
20      fprintf(stderr, "%s: DNS error %d\n", msg, h_errno);
21      exit(0);
22  }
23
24  void app_error(char *msg) /* application error */
25  {
26      fprintf(stderr, "%s\n", msg);
27      exit(0);
```

```
28  }
29
30  /**********************************************
31   * Wrappers for Unix process control functions
32   **********************************************/
33
34  pid_t Fork(void)
35  {
36      pid_t pid;
37
38      if ((pid = fork()) < 0)
39          unix_error("Fork error");
40      return pid;
41  }
42
43  void Execve(const char *filename, char *const argv[], char *const envp[])
44  {
45      if (execve(filename, argv, envp) < 0)
46          unix_error("Execve error");
47  }
48
49  pid_t Wait(int *status)
50  {
51      pid_t pid;
52
53      if ((pid = wait(status)) < 0)
54          unix_error("Wait error");
55      return pid;
56  }
57
58  pid_t Waitpid(pid_t pid, int *iptr, int options)
59  {
60      pid_t retpid;
61
62      if ((retpid = waitpid(pid, iptr, options)) < 0)
63          unix_error("Waitpid error");
64      return(retpid);
65  }
66
67  void Kill(pid_t pid, int signum)
68  {
69      int rc;
70
71      if ((rc = kill(pid, signum)) < 0)
72          unix_error("Kill error");
```

```
73  }
74
75  void Pause()
76  {
77      (void)pause();
78      return;
79  }
80
81  unsigned int Sleep(unsigned int secs)
82  {
83      unsigned int rc;
84
85      if ((rc = sleep(secs)) < 0)
86          unix_error("Sleep error");
87      return rc;
88  }
89
90  unsigned int Alarm(unsigned int seconds) {
91      return alarm(seconds);
92  }
93
94  void Setpgid(pid_t pid, pid_t pgid) {
95      int rc;
96
97      if ((rc = setpgid(pid, pgid)) < 0)
98          unix_error("Setpgid error");
99      return;
100 }
101
102 pid_t Getpgrp(void) {
103     return getpgrp();
104 }
105
106 /************************************
107  * Wrappers for Unix signal functions
108  ***********************************/
109
110 handler_t *Signal(int signum, handler_t *handler)
111 {
112     struct sigaction action, old_action;
113
114     action.sa_handler = handler;
115     sigemptyset(&action.sa_mask); /* block sigs of type being handled */
116     action.sa_flags = SA_RESTART; /* restart syscalls if possible */
117
```

```
118     if (sigaction(signum, &action, &old_action) < 0)
119         unix_error("Signal error");
120     return (old_action.sa_handler);
121 }
122
123 void Sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
124 {
125     if (sigprocmask(how, set, oldset) < 0)
126         unix_error("Sigprocmask error");
127     return;
128 }
129
130 void Sigemptyset(sigset_t *set)
131 {
132     if (sigemptyset(set) < 0)
133         unix_error("Sigemptyset error");
134     return;
135 }
136
137 void Sigfillset(sigset_t *set)
138 {
139     if (sigfillset(set) < 0)
140         unix_error("Sigfillset error");
141     return;
142 }
143
144 void Sigaddset(sigset_t *set, int signum)
145 {
146     if (sigaddset(set, signum) < 0)
147         unix_error("Sigaddset error");
148     return;
149 }
150
151 void Sigdelset(sigset_t *set, int signum)
152 {
153     if (sigdelset(set, signum) < 0)
154         unix_error("Sigdelset error");
155     return;
156 }
157
158 int Sigismember(const sigset_t *set, int signum)
159 {
160     int rc;
161     if ((rc = sigismember(set, signum)) < 0)
162         unix_error("Sigismember error");
```

```
163     return rc;
164 }
165
166
167 /*******************************
168  * Wrappers for Unix I/O routines
169  ********************************/
170
171 int Open(const char *pathname, int flags, mode_t mode)
172 {
173     int rc;
174
175     if ((rc = open(pathname, flags, mode)) < 0)
176         unix_error("Open error");
177     return rc;
178 }
179
180 ssize_t Read(int fd, void *buf, size_t count)
181 {
182     ssize_t rc;
183
184     if ((rc = read(fd, buf, count)) < 0)
185         unix_error("Read error");
186     return rc;
187 }
188
189 ssize_t Write(int fd, const void *buf, size_t count)
190 {
191     ssize_t rc;
192
193     if ((rc = write(fd, buf, count)) < 0)
194         unix_error("Write error");
195     return rc;
196 }
197
198 off_t Lseek(int fildes, off_t offset, int whence)
199 {
200     off_t rc;
201
202     if ((rc = lseek(fildes, offset, whence)) < 0)
203         unix_error("Lseek error");
204     return rc;
205 }
206
207 void Close(int fd)
```

```
208 {
209     int rc;
210 
211     if ((rc = close(fd)) < 0)
212         unix_error("Close error");
213 }
214 
215 int Select(int n, fd_set *readfds, fd_set *writefds,
216            fd_set *exceptfds, struct timeval *timeout)
217 {
218     int rc;
219 
220     if ((rc = select(n, readfds, writefds, exceptfds, timeout)) < 0)
221         unix_error("Select error");
222     return rc;
223 }
224 
225 int Dup2(int fd1, int fd2)
226 {
227     int rc;
228 
229     if ((rc = dup2(fd1, fd2)) < 0)
230         unix_error("Dup2 error");
231     return rc;
232 }
233 
234 void Stat(const char *filename, struct stat *buf)
235 {
236     if (stat(filename, buf) < 0)
237         unix_error("Stat error");
238 }
239 
240 void Fstat(int fd, struct stat *buf)
241 {
242     if (fstat(fd, buf) < 0)
243         unix_error("Fstat error");
244 }
245 
246 /****************************************
247  * Wrappers for memory mapping functions
248  ****************************************/
249 void *Mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
250 {
251     void *ptr;
252 
```

```
253     if ((ptr = mmap(addr, len, prot, flags, fd, offset)) == ((void *) -1))
254         unix_error("mmap error");
255     return(ptr);
256 }
257 
258 void Munmap(void *start, size_t length)
259 {
260     if (munmap(start, length) < 0)
261         unix_error("munmap error");
262 }
263 
264 /***************************************************
265  * Wrappers for dynamic storage allocation functions
266  ***************************************************/
267 
268 void *Malloc(size_t size)
269 {
270     void *p;
271 
272     if ((p = malloc(size)) == NULL)
273         unix_error("Malloc error");
274     return p;
275 }
276 
277 void *Realloc(void *ptr, size_t size)
278 {
279     void *p;
280 
281     if ((p = realloc(ptr, size)) == NULL)
282         unix_error("Realloc error");
283     return p;
284 }
285 
286 void *Calloc(size_t nmemb, size_t size)
287 {
288     void *p;
289 
290     if ((p = calloc(nmemb, size)) == NULL)
291         unix_error("Calloc error");
292     return p;
293 }
294 
295 void Free(void *ptr)
296 {
297     free(ptr);
```

```
298 }
299
300 /******************************************
301  * Wrappers for the Standard I/O functions.
302  ******************************************/
303 void Fclose(FILE *fp)
304 {
305     if (fclose(fp) != 0)
306         unix_error("Fclose error");
307 }
308
309 FILE *Fdopen(int fd, const char *type)
310 {
311     FILE *fp;
312
313     if ((fp = fdopen(fd, type)) == NULL)
314         unix_error("Fdopen error");
315
316     return fp;
317 }
318
319 char *Fgets(char *ptr, int n, FILE *stream)
320 {
321     char *rptr;
322
323     if (((rptr = fgets(ptr, n, stream)) == NULL) && ferror(stream))
324         app_error("Fgets error");
325
326     return rptr;
327 }
328
329 FILE *Fopen(const char *filename, const char *mode)
330 {
331     FILE *fp;
332
333     if ((fp = fopen(filename, mode)) == NULL)
334         unix_error("Fopen error");
335
336     return fp;
337 }
338
339 void Fputs(const char *ptr, FILE *stream)
340 {
341     if (fputs(ptr, stream) == EOF)
342         unix_error("Fputs error");
```

```
343 }
344
345 size_t Fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
346 {
347     size_t n;
348
349     if (((n = fread(ptr, size, nmemb, stream)) < nmemb) && ferror(stream))
350         unix_error("Fread error");
351     return n;
352 }
353
354 void Fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
355 {
356     if (fwrite(ptr, size, nmemb, stream) < nmemb)
357         unix_error("Fwrite error");
358 }
359
360
361 /****************************
362  * Sockets interface wrappers
363  ****************************/
364
365 int Socket(int domain, int type, int protocol)
366 {
367     int rc;
368
369     if ((rc = socket(domain, type, protocol)) < 0)
370         unix_error("Socket error");
371     return rc;
372 }
373
374 void Setsockopt(int s, int level, int optname, const void *optval, int optlen)
375 {
376     int rc;
377
378     if ((rc = setsockopt(s, level, optname, optval, optlen)) < 0)
379         unix_error("Setsockopt error");
380 }
381
382 void Bind(int sockfd, struct sockaddr *my_addr, int addrlen)
383 {
384     int rc;
385
386     if ((rc = bind(sockfd, my_addr, addrlen)) < 0)
387         unix_error("Bind error");
```

```
388 }
389
390 void Listen(int s, int backlog)
391 {
392     int rc;
393
394     if ((rc = listen(s, backlog)) < 0)
395         unix_error("Listen error");
396 }
397
398 int Accept(int s, struct sockaddr *addr, int *addrlen)
399 {
400     int rc;
401
402     if ((rc = accept(s, addr, addrlen)) < 0)
403         unix_error("Accept error");
404     return rc;
405 }
406
407 void Connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
408 {
409     int rc;
410
411     if ((rc = connect(sockfd, serv_addr, addrlen)) < 0)
412         unix_error("Connect error");
413 }
414
415 /************************
416  * DNS interface wrappers
417  ***********************/
418
419 struct hostent *Gethostbyname(const char *name)
420 {
421     struct hostent *p;
422
423     if ((p = gethostbyname(name)) == NULL)
424         dns_error("Gethostbyname error");
425     return p;
426 }
427
428 struct hostent *Gethostbyaddr(const char *addr, int len, int type)
429 {
430     struct hostent *p;
431
432     if ((p = gethostbyaddr(addr, len, type)) == NULL)
```

```
433            dns_error("Gethostbyaddr error");
434        return p;
435    }
436
437    /***********************************************
438     * Wrappers for Pthreads thread control functions
439     ***********************************************/
440
441    void Pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
442                        void * (*routine)(void *), void *argp)
443    {
444        int rc;
445
446        if ((rc = pthread_create(tidp, attrp, routine, argp)) != 0)
447            posix_error(rc, "Pthread_create error");
448    }
449
450    void Pthread_cancel(pthread_t tid) {
451        int rc;
452
453        if ((rc = pthread_cancel(tid)) != 0)
454            posix_error(rc, "Pthread_cancel error");
455    }
456
457    void Pthread_join(pthread_t tid, void **thread_return) {
458        int rc;
459
460        if ((rc = pthread_join(tid, thread_return)) != 0)
461            posix_error(rc, "Pthread_join error");
462    }
463
464    void Pthread_detach(pthread_t tid) {
465        int rc;
466
467        if ((rc = pthread_detach(tid)) != 0)
468            posix_error(rc, "Pthread_detach error");
469    }
470
471    void Pthread_exit(void *retval) {
472        pthread_exit(retval);
473    }
474
475    pthread_t Pthread_self(void) {
476        return pthread_self();
477    }
```

```
478
479 void Pthread_once(pthread_once_t *once_control, void (*init_function)()) {
480     pthread_once(once_control, init_function);
481 }
482
483 /*******************************
484  * Wrappers for Posix semaphores
485  *******************************/
486
487 void Sem_init(sem_t *sem, int pshared, unsigned int value)
488 {
489     if (sem_init(sem, pshared, value) < 0)
490         unix_error("Sem_init error");
491 }
492
493 void P(sem_t *sem)
494 {
495     if (sem_wait(sem) < 0)
496         unix_error("P error");
497 }
498
499 void V(sem_t *sem)
500 {
501     if (sem_post(sem) < 0)
502         unix_error("V error");
503 }
504
505 /*********************************************************
506  * The Rio package - robust I/O functions
507  *********************************************************/
508 /*
509  * rio_readn - robustly read n bytes (unbuffered)
510  */
511 ssize_t rio_readn(int fd, void *usrbuf, size_t n)
512 {
513     size_t nleft = n;
514     ssize_t nread;
515     char *bufp = usrbuf;
516
517     while (nleft > 0) {
518         if ((nread = read(fd, bufp, nleft)) < 0) {
519             if (errno == EINTR) /* interrupted by sig handler return */
520                 nread = 0;      /* and call read() again */
521             else
522                 return -1;      /* errno set by read() */
```

```
523             }
524             else if (nread == 0)
525                 break;              /* EOF */
526             nleft -= nread;
527             bufp += nread;
528         }
529         return (n - nleft);         /* return >= 0 */
530     }
531 
532     /*
533      * rio_writen - robustly write n bytes (unbuffered)
534      */
535     ssize_t rio_writen(int fd, void *usrbuf, size_t n)
536     {
537         size_t nleft = n;
538         ssize_t nwritten;
539         char *bufp = usrbuf;
540 
541         while (nleft > 0) {
542             if ((nwritten = write(fd, bufp, nleft)) <= 0) {
543                 if (errno == EINTR)  /* interrupted by sig handler return */
544                     nwritten =       /* and call write() again */
545                 else
546                     return -1;       /* errorno set by write() */
547             }
548             nleft -= nwritten;
549             bufp += nwritten;
550         }
551         return n;
552     }
553 
554 
555     /*
556      * rio_read - This is a wrapper for the Unix read() function that
557      * transfers min(n, rio_cnt) bytes from an internal buffer to a user
558      * buffer, where n is the number of bytes requested by the user and
559      * rio_cnt is the number of unread bytes in the internal buffer. On
560      * entry, rio_read() refills the internal buffer via a call to
561      * read() if the internal buffer is empty.
562      */
563     static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
564     {
565         int cnt;
566 
567      while (rp->rio_cnt <= 0) {  /* refill if buf is empty */
```

```
568         rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
569                            sizeof(rp->rio_buf));
570         if (rp->rio_cnt < 0) {
571             if (errno != EINTR) /* interrupted by sig handler return */
572                 return -1;
573         }
574         else if (rp->rio_cnt == 0) /* EOF */
575             return 0;
576         else
577             rp->rio_bufptr = rp->rio_buf; /* reset buffer ptr */
578     }
579 
580     /* Copy min(n, rp->rio_cnt) bytes from internal buf to user buf */
581     cnt = n;
582     if (rp->rio_cnt < n)
583         cnt = rp->rio_cnt;
584     memcpy(usrbuf, rp->rio_bufptr, cnt);
585     rp->rio_bufptr += cnt;
586     rp->rio_cnt -= cnt;
587     return cnt;
588 }
589 
590 /*
591  * rio_readinitb - Associate a descriptor with a read buffer and reset buffer
592  */
593 void rio_readinitb(rio_t *rp, int fd)
594 {
595     rp->rio_fd = fd;
596     rp->rio_cnt = 0;
597     rp->rio_bufptr = rp->rio_buf;
598 }
599 
600 /*
601  * rio_readnb - Robustly read n bytes (buffered)
602  */
603 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
604 {
605     size_t nleft = n;
606     ssize_t nread;
607     char *bufp = usrbuf;
608 
609     while (nleft > 0) {
610         if ((nread = rio_read(rp, bufp, nleft)) < 0) {
611             if (errno == EINTR)     /* interrupted by sig handler return */
612                 nread = 0;          /* call read() again */
```

```
613             else
614                 return -1;           /* errno set by read() */
615         }
616         else if (nread == 0)
617             break;          /* EOF */
618         nleft -= nread;
619         bufp += nread;
620     }
621     return (n - nleft);    /* return >= 0 */
622 }
623
624 /*
625  * rio_readlineb - robustly read a text line (buffered)
626  */
627 ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
628 {
629     int n, rc;
630     char c, *bufp = usrbuf;
631
632     for (n = 1; n < maxlen; n++) {
633         if ((rc = rio_read(rp, &c, 1)) == 1) {
634             *bufp++ = c;
635             if (c == '\n')
636                 break;
637         } else if (rc == 0) {
638             if (n == 1)
639                 return 0;   /* EOF, no data read */
640           else
641               break;        /* EOF, some data was read */
642         } else
643             return -1;      /* error */
644     }
645     *bufp = 0;
646     return n;
647 }
648
649 /**********************************
650  * Wrappers for robust I/O routines
651  **********************************/
652 ssize_t Rio_readn(int fd, void *ptr, size_t nbytes)
653 {
654     ssize_t n;
655
656     if ((n = rio_readn(fd, ptr, nbytes)) < 0)
657         unix_error("Rio_readn error");
```

```
658     return n;
659 }
660
661 void Rio_writen(int fd, void *usrbuf, size_t n)
662 {
663     if (rio_writen(fd, usrbuf, n) != n)
664         unix_error("Rio_writenb error");
665 }
666
667 void Rio_readinitb(rio_t *rp, int fd)
668 {
669     rio_readinitb(rp, fd);
670 }
671
672 ssize_t Rio_readnb(rio_t *rp, void *usrbuf, size_t n)
673 {
674     ssize_t rc;
675
676     if ((rc = rio_readnb(rp, usrbuf, n)) < 0)
677         unix_error("Rio_readnb error");
678     return rc;
679 }
680
681 ssize_t Rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
682 {
683     ssize_t rc;
684
685     if ((rc = rio_readlineb(rp, usrbuf, maxlen)) < 0)
686         unix_error("Rio_readlineb error");
687     return rc;
688 }
689
690 /********************************
691  * Client/server helper functions
692  ********************************/
693 /*
694  * open_clientfd - open connection to server at <hostname, port>
695  *     and return a socket descriptor ready for reading and writing.
696  *     Returns -1 and sets errno on Unix error.
697  *     Returns -2 and sets h_errno on DNS (gethostbyname) error.
698  */
699 int open_clientfd(char *hostname, int port)
700 {
701     int clientfd;
702     struct hostent *hp;
```

```
703         struct sockaddr_in serveraddr;
704
705         if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
706             return -1; /* check errno for cause of error */
707
708         /* Fill in the server's IP address and port */
709         if ((hp = gethostbyname(hostname)) == NULL)
710             return -2; /* check h_errno for cause of error */
711         bzero((char *) &serveraddr, sizeof(serveraddr));
712         serveraddr.sin_family = AF_INET;
713         bcopy((char *)hp->h_addr,
714               (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
715         serveraddr.sin_port = htons(port);
716
717         /* Establish a connection with the server */
718         if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
719            return -1;
720         return clientfd;
721     }
722
723     /*
724      * open_listenfd - open and return a listening socket on port
725      *         Returns -1 and sets errno on Unix error.
726      */
727     int open_listenfd(int port)
728     {
729         int listenfd, optval=1;
730         struct sockaddr_in serveraddr;
731
732         /* Create a socket descriptor */
733         if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
734             return -1;
735
736         /* Eliminates "Address already in use" error from bind. */
737         if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
738                        (const void *)&optval , sizeof(int)) < 0)
739             return -1;
740
741        /* Listenfd will be an endpoint for all requests to port
742             on any IP address for this host */
743        bzero((char *) &serveraddr, sizeof(serveraddr));
744        serveraddr.sin_family = AF_INET;
745        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
746        serveraddr.sin_port = htons((unsigned short)port);
747        if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
```

```
748             return -1;
749 
750         /* Make it a listening socket ready to accept connection requests */
751         if (listen(listenfd, LISTENQ) < 0)
752             return -1;
753         return listenfd;
754     }
755 
756     /***********************************************
757      * Wrappers for the client/server helper routines
758      ***********************************************/
759     int Open_clientfd(char *hostname, int port)
760     {
761         int rc;
762 
763         if ((rc = open_clientfd(hostname, port)) < 0) {
764             if (rc == -1)
765                 unix_error("Open_clientfd Unix error");
766             else
767                 dns_error("Open_clientfd DNS error");
768         }
769         return rc;
770     }
771 
772     int Open_listenfd(int port)
773     {
774         int rc;
775 
776         if ((rc = open_listenfd(port)) < 0)
777             unix_error("Open_listenfd error");
778         return rc;
779     }
```

code/src/csapp.c